Identification and Management of Inconsistencies in Dynamically Adaptive Software Systems

ICTEAM Institute
École Polytechnique de Louvain
Université catholique de Louvain

Software Languages Lab
Faculteit van de Wetenschappen
Vrije Universiteit Brussel

Dissertation

Identification and Management of Inconsistencies in Dynamically Adaptive Software Systems

Nicolás Cardozo Álvarez

3rd September 2013

Thesis submitted in partial fulfillment of the requirements for the degree of Doctor in Engineering Sciences at Université catholique de Louvain and Doctor in Sciences at Vrije Universiteit Brussel

Thesis Committee:

Prof. Kim Mens (Promotor)	UCL, Belgium
Prof. Theo D'Hondt (Promotor)	VUB, Belgium
Prof. Charles Pecheur (Chair)	UCL, Belgium
Prof. Tom Van Cutsem (Secretary)	VUB, Belgium
Prof. Hidehiko Masuhara	Tokyo Institute of Technology, Japan
Prof. Gilles Geeraerts	ULB, Belgium
Prof. Jean Vanderdonckt	UCL, Belgium
Prof. Bernard Manderick	VUB, Belgium

Identification and Management of Inconsistencies in Dynamically Adaptive Software Systems

ICTEAM Institute
École Polytechnique de Louvain
Université catholique de Louvain
Place Sainte-Barbe, 2
1348 Louvain-la-Neuve
Belgium

Software Languages Lab
Faculteit van de Wetenschappen
Vrije Universiteit Brussel
Pleinlaan, 2
1050 Brussel
Belgium

This work has been supported by the VariBru project of the ICT Impulse Programme of the Brussels Institute for Innovation and Research (INNOVIRIS).

This manuscript has been set with the help of TEXSHOP and PDFLATEX (with BIBTEX support). Bugs have been tracked down in the text and squashed thanks to *Bugs in Writing* by Dupré [57] and *Elements of Style* by Strunk and White [180].

To *me, myself and I.*

Abstract

Computing devices now enable access to rich information about their surrounding execution environment gathered through sensor networks or system monitors. This ability allows software systems to be conceived with context in mind, instead of being created in isolation as in traditional approaches for software development. Services provided by software systems can be adapted to sensed conditions rendering such services more appropriate to the surrounding execution environment. Adaptations to the system's behavior take place unannounced over time. However, if not dealt with carefully, the behavior provided by such adaptations could lead to inconsistencies in the system's behavior. In order to avoid such inconsistencies, dependencies between adaptations must be carefully managed so that interactions gathered for the surrounding execution environment are not rendered incompatible.

This dissertation investigates how to provide more guarantees about the predictability of the system's behavior when it is adapted dynamically at run time. Based on the observation of different dynamically adaptive software systems, we put forward a set of requirements that software systems should satisfy to ensure consistency of its behavioral adaptations. We propose a formal basis to support the development of consistent software systems in the presence of dynamic behavioral adaptations, called context Petri nets. This formal basis complies with the requirements for consistent dynamically adaptive software systems, and in particular context-oriented programming, on three levels: formalization, execution, and analysis.

Context Petri nets offer a formalization for the definition of adaptations, the interactions between them, and the notion of consistency of a system in the presence of dynamic behavioral adaptations. Interactions between adaptations are formalized by a well-defined set of rules that capture the intention of programmers at a high-level, while enabling the low-level representation and automatic verification of those rules. Consistency verification of the system is provided at two levels. At design-time, system properties can be analyzed for the identification of possible incoherence in the definition of interactions between adaptations. At run-time the satisfiability of all interaction rules between adaptations is verified, hence, it can be ensured that no inconsistencies occur. Based on the proposed formal basis, we offer a tool for the design, manipulation, and simulation of adaptations and their interactions. This work is validated by demonstrating its usefulness in analyzing existing context-aware applications, its appropriateness in broadening the frontiers of context-oriented programming, and its extensibility by expanding the formal basis itself.

Samenvatting

Rekenkrachtige apparaten hebben tegenwoordig toegang tot een rijke hoeveelheid aan informatie over hun omliggende omgeving, waargenomen door netwerksensoren en systeemmonitors. Dit laat toe om software-systemen te ontwerpen die zich aanpassen aan hun context, in tegenstelling tot meer traditionele systemen die veelal worden ontwikkeld in isolatie van omgevingsfactoren. Diensten aangeboden door dergelijke software-systemen kunnen zich aanpassen aan de waargenomen omstandigheden, waardoor deze diensten beter aangepast zijn aan de omliggende omgeving. Aanpassingen aan het gedrag van het systeem kunnen onaangekondigd plaatsvinden op om het even welk ogenblik. Indien niet met de nodige omzichtigheid wordt opgetreden, kunnen dergelijke aanpassingen daarom leiden tot inconsistenties in het gedrag van het systeem. Om dergelijke inconsistenties te vermijden, moeten afhankelijkheden tussen mogelijke aanpassingen degelijk gedocumenteerd en zorgvuldig beheerd worden, om er aldus voor te zorgen dat de mogelijke interacties tussen deze aanpassingen niet onverenigbaar zijn voor de omliggende omgevingsfactoren.

Dit proefschrift onderzoekt welke garanties kunnen geboden worden omtrent de voorspelbaarheid van het gedrag van software-systemen die dynamisch kunnen aangepast worden tijdens hun uitvoering. Op basis van een studie van verschillende dynamisch adaptieve software-systemen stellen wij de nodige vereisten op waaraan software-systemen moeten voldoen om de consistentie van gedragsaanpassingen van dergelijke systemen te garanderen. We stellen een formele basis voor, genaamd contextuele Petri netten, ter ondersteuning van de ontwikkeling van consistente softwaresystemen in de aanwezigheid van dynamische gedragsaanpassingen. Deze formele basis voldoet aan de eerder opgestelde vereisten voor consistente dynamisch adaptieve software-systemen, en context georiënteerde programma's in het bijzonder, op drie niveaus: formalisering, uitvoering en analyse.

Contextuele Petri netten bieden een formalisering aan voor de definitie van dynamische gedragsaanpassingen, de interactie daartussen, en de notie van een consistent systeem in de aanwezigheid van dergelijke dynamische gedragsaanpassingen. Interacties tussen aanpassingen worden geformaliseerd door welbepaalde regels die enerzijds de intentie van een programmeur op een hoog niveau weten te vatten, en anderzijds een representatie op laag niveau bieden die een automatische verificatie van de regels toelaten. Verificatie van de consistentie van een dynamisch adaptief software-systeem kan dan worden voorzien op twee niveaus. Enerzijds kunnen tijdens het ontwerp en de ontwikkeling van het systeem bepaalde systeemeigenschappen geanalyseerd worden, om in een vroeg stadium mogelijke incoherenties in de definitie van interacties tussen aanpassin-

gen te identificeren. Anderzijds kunnen tijdens de uitvoering van een dergelijk systeem de gedefinieerde interactieregels tussen aanpassingen continue geverifieerd worden om aldus te garanderen dat er zich geen inconsistenties zullen voordoen. Met deze voorgestelde formele basis als fundament bieden we een hulpmiddel aan voor het ontwerp, het beheer de de simulatie van dynamische aanpassingen en hun interacties.

Dit werk wordt gevalideerd door het nut aan te tonen van de aanpak bij het analyseren van bestaande context georiënteerde toepassingen, door de grenzen van context georiënteerd programmeren te verruimen, en door de uitbreidbaarheid van de formele basis te illustreren.

Acknowledgements

Thinking back on who to thank in the acknowledgements
I realized how irresponsible would it be to make one single
person accountable for the last four years of my life
..... that being said .....

The story of this dissertation can be traced back to early 2007, when a very stubborn professor tried to convince me to do my master degree in France. What a ridiculous idea. Me in France? Noooooo way. And look at me now, five years later I have to eat all of my words (some of them in actual french), solely because of Rubby Casallas. She was the first one to believe. She sent me here and is the one that started this whole adventure. She is the one to blame for what became of me.

When I first started the master, it was never part of my plan to do a Ph.D., I had considered it before, but it was never the plan. Let alone could I ever imagine in my wildest dreams that at some point I would end up in Belgium. Belgium of all places!! In retrospective I guess it all worked for the best. All I ever wanted to do with the master was partying like I didn't during my bachelor degree. And it worked... for like 1 week until a lot of eye-opening topics from my professors during the master (thanks Wolf, Jorge, Mario Südholt) made me the geeky kid that I am now. The master was finished only thanks to the super-human patience that both Elisa and Jorge had not to kill me while writing my master thesis. Even more it was thanks to Elisa and Jorge that I got the extra push to move to the UCL to start this dissertation, a big THANK YOU guys.

The following four years of my life went as expected. Everything that you hear about a Ph.D.s is true, it is enslaving, difficult, self-confronting, and lonely. Odds are against you to finish it. What can really make the difference is who your enslaver is, and who is slaving next to you. Luckily, I won the lottery in both regards. As far as masters go, Kim has to be the coolest (to the point that slavery would still exists if people back then would be a tenth of cool and overall good people as he is). Kim took me under his wing without ever meeting me, and yes he started regretting it from day two (don't worry Kim your martyrdom is now over), but anyways kept on helping/pushing me to find a topic, write papers, be critical, and basically become a better me. I am going to miss getting your mails at 1 in the morning. What Kim did out of me was not an easy job (my family can tell you all about it). It was hard, but it all worked out fine. I will always remember that tipping point of approval that read "*I never thought I would say this but Petri nets are kind of cool*". Those are the most beautiful words that anyone has ever tweeted (not) to me. Of course Kim didn't do it all alone (not even him could do it alone), everyone in

RELEASeD (Angela, Sebas, Diego, Alfredo, Sergio, Johan) did help a lot, and the outcome of this work is as much theirs as it is mine (although it actually belongs to **Chuck Norris**,[1] after all it was his idea).

It took one short year for everyone at UCL to get sick of me, so they shipped me out back to VUB. Luckily for me almost any other lab is worse to end up at than Soft (and the many past and present people there). Theo took me under his wing as promotor and with the help of Pascal, Sebastian Günther, Jorge, and Ragnhild we got through the best and worse of the following years. *I still think is serendipity that ALL of them moved away after a couple of months of interacting with me, but hey...history will tell.* To everyone at Soft and RELEASeD, but specially Kim, Theo and Sebas, I owe my biggest thanks, for crushing my bad ideas, critically encouraging the ok ones and helping me out with whatever I needed help with to reach this point. I truly can't think of the words to express my gratitude, so I'm not even going to try, in any case you guys can probably think about more eloquent ways of saying it than me.

Over the past four years I had many and long discussions with countless people (mainly Kim, Sebas, Theo, Ragnhild, Jorge, Sebastian, Pascal, god). Discussions that shaped and enlightened my work. I have to say that, although it may not show, I did take into account every piece of advice that everybody gave me from every research presentation, informal discussions, or drunken rambles. To Sebas I owe everything, he was there from the very first to the very last. I think it has never been so enriching to talk to/across a screen.

The text that follows is a nice combination of comments and help of many people. In first place there is my very crowded jury, Gilles, Hidehiko, Charles, Bernard, Tom (thanks for stepping up to the plate last minute, specially with the secretary work), and Jean, and obviously Kim and Theo (although they had to be there). Thanks for your comments on earlier version of this text, it really help to improve it. To Theo, Kim, Eline, Sebas and Ragnhild who helped me revising earlier versions of the text I can only say that I'm sorry, I'm really really sorry, I'm also grateful and all that, don't get me wrong, but mainly I am very sorry. The last couple of months towards finishing the text were not easy and any trace of quality that this text may have is all because of you guys. I can actually say the same for the presentation, which went from complete disaster to nice looking and content-full. Everyone that went to my many many dry-runs, (I remember) Angela, Sebas, Kim, Theo, Elisa, Christophe, Stefan, Coen, Wolf, Niels, Eline, Ragnhild, really improved the presentation a lot.

This is kind of it. That is the story of how everything came to be and we all ended up sitting here reading this text. As you may have noticed by now, the acknowledgements have been work-related, so it is not by chance that you haven't found your name yet (sorry for wasting your time this far). Unfortunately, fun parties and family dinners are out of the scope of this dissertation and thus we do not mention them in extend in this text. It might strike you as a shock, but I did managed to maintain a more or less real life during this period. I obviously did not do it on my own. Enough cool people in the world

[1] By law every mention of **Chuck Norris** has to be in bold.

kind of like me, which is alright I guess. This is the part where I thank my family, they were always there by my side, and yes they had to, because... you know...family and all that, but I kind'a think they want to anyways. To all my friends, from here and there, thank you guys, you are awesome, a lot of fun and I really really like you. You guys know who you are..... and in case you don't.....

m	a	m	i	w	w	j	o	k	l	r	e	e	n	i	g	n	e	b	o	k
f	a	q	x	m	k	e	n	n	e	d	y	z	j	d	l	j	c	e	c	l
r	g	t	o	m	v	a	n	c	u	t	s	e	m	s	n	p	a	o	s	a
a	n	n	b	e	d	i	v	a	d	e	s	l	a	p	w	u	r	i	f	u
n	n	j	l	h	b	e	r	t	c	s	c	a	r	o	l	z	m	o	m	r
l	y	i	r	p	o	l	n	e	i	f	g	z	i	l	e	o	l	o	d	e
e	n	i	e	l	s	o	l	e	v	x	k	n	o	o	n	k	h	r	i	n
e	n	m	i	n	h	**c**	**h**	**u**	**c**	**k**	**n**	**o**	**r**	**r**	**i**	**s**	c	r	e	t
m	n	e	s	d	j	n	a	h	o	j	z	g	h	a	n	s	a	y	r	c
a	v	j	o	e	r	i	x	r	o	w	p	a	n	g	u	n	r	j	i	h
a	t	e	r	c	l	u	t	r	l	u	n	s	e	v	y	e	g	e	c	r
r	h	b	a	m	b	i	g	s	g	o	e	i	f	a	r	m	i	z	k	i
t	i	d	i	d	i	e	r	n	i	n	s	l	v	r	i	m	b	i	x	s
e	e	l	b	d	m	d	i	d	i	e	r	e	n	e	m	i	e	p	r	t
n	r	f	u	t	a	t	w	l	k	n	a	r	f	u	k	k	o	a	l	o
v	r	r	r	e	t	u	e	m	e	d	g	n	a	g	f	l	o	w	o	p
a	y	e	p	e	t	e	r	s	k	a	t	t	u	o	n	i	e	r	e	h
n	u	v	s	e	b	a	s	t	i	a	n	g	o	n	z	a	l	e	z	e
i	j	y	a	l	o	c	i	n	s	n	e	j	r	s	b	d	r	i	e	s
n	u	u	m	i	a	n	d	y	d	i	e	g	o	o	o	l	y	s	y	c
g	e	g	j	o	p	a	r	i	c	i	a	k	i	l	s	d	y	e	t	h
e	e	a	r	a	g	n	h	i	l	d	m	l	h	r	n	f	r	r	t	o
l	a	u	r	e	p	h	i	l	i	p	s	t	r	a	r	p	e	d	o	l
g	z	i	d	a	c	a	e	z	c	m	u	o	s	c	a	r	r	n	c	l
e	l	e	i	j	t	v	s	l	e	i	h	c	i	m	l	e	b	a	s	i
m	a	n	t	i	a	s	d	e	w	a	e	l	e	p	f	n	e	l	l	e
c	o	n	v	i	v	i	a	n	e	j	o	n	c	k	e	r	s	h	a	r
s	m	a	m	u	e	l	a	w	o	r	t	s	a	c	o	i	g	r	e	s
m	a	s	y	d	n	i	c	w	o	r	t	s	c	h	e	p	e	r	s	a

Not to close this on a bummer note, but I remember how much people told me about all the times I was going to hate myself while doing the Ph.D., be depress and want to throw it all out the board. They were kind of wright, I had many of those moments (it is meant to be). Such moments never resonated with me for one simple reason. One of the last things that my father ever told me, waaaaay before the fact, was that I seem convinced about the path I wanted to follow in life, and that this was the right decision for me. I was surely going to make it. So all of those moments when I wasn't feeling in the right place or just felt plain out homesick I remembered what he told me, and just repeating his words in my head would make everything ok again. I don't know about you, but I would like to think that it is because of my dad that I made it this far.[2]

[2]The dedication of this thesis is meant as a joke, it has a nice catchy sound to it, that is all. I am not a douchebag, this thesis is dedicated to you.

Contents

List of Figures

Chapter 1

Introduction

1.1 Research Context

Over the last couple of decades software systems have come to pervade our daily lives. Software systems can be found everywhere in our surroundings, on desktop computers, mobile devices, on-board systems, household appliances, and in intelligent city environments. Software systems are spreading out as utilities to aid in our daily lives, where they are rapidly becoming more ubiquitous. That is, software systems are gaining the capacity to perform and manage their tasks without requiring user interaction or supervision; robo-train subway systems and self-balancing web servers are a case in point. As a matter of fact, following the vision of ubiquitous computing [193], software systems are able to gather information about their surrounding environment, and use it to adapt their behavior.

Software development is shifting from systems conceived in isolation to systems that are aware of, and interact seamlessly with their environment. Such interactive software systems are able to provide their users with *smarter* services, while remaining *oblivious* to them. Systems become smarter thanks to their awareness of their surroundings. The services provided by such software systems are not one-size-fits-all services. Rather, they can use the information about their **surrounding execution environment** —that is, the internal and external conditions of the system at run time. This information is made available at any moment in time, to provide "tailor-made" services that are deemed more appropriate. For example, a system could provide optimized algorithms when additional computing resources are available in the surrounding execution environment, or delegated services when less resources become available. Current-day software systems can easily gather such information, thanks to the proliferation of hardware devices equipped with a variety of sensors able to constantly retrieve information about their surrounding environment. Software systems are now able to access rich information about their surrounding environment. Changes to the services offered by such system are not required to be explicitly introduced by users. Instead software systems can be refined

automatically, expanding the frontiers of such systems.

The work presented hereinafter proposes a formal basis for the development of dynamically adaptive software systems, presented for the particular case of Context-Oriented Programming; that is, we focus on software systems that are able to interact with their surrounding execution environment and appropriately adapt their behavior at run time. The particular objective of developing such a formal basis, is to enable the analysis and management of such software systems, in order to prevent inconsistent behavior in the presence of dynamic adaptations.

Different approaches have been proposed to enable the dynamic adaptivity of the behavior of a software system [117, 152, 149]. In particular, over the last few years there has been a particular increase in the development of programming languages that enable a class of highly dynamic software systems, allowing behavioral adaptations with respect to their surrounding execution environment [44, 74, 185, 103, 163, 9]. This class of systems, known as Context-Oriented Programming (COP) systems, envision highly dynamic environments where the conditions under which software systems execute change constantly and, hence, influence the behavior of the system. Information about the surrounding execution environment of the system is usually enabled via sensor networks or system monitors. Example situations for which a system can have behavioral adaptations using the information gathered from its surrounding execution environment include: the global positioning of a user gathered through a GPS antenna, the amount of light in a room gathered through an integrated luminosity sensor, the number of idle cores in a multi-core system gathered through an active monitor, the language displayed on a user interface gathered through user-defined preferences, or the availability of external services gathered through a service discovery monitor.

Highly dynamic and adaptive software systems, as proposed by COP open the possibility to new application domains, new opportunities to extend software services functionality, and to improve their quality. Software systems now have the possibility of defining dedicated behavior, objects, services, or properties specific to particular situations of their surrounding execution environment, turning such systems into constantly evolving software systems. We notice, however, that these type of systems do not truly exist yet. In a truly open environment, where software systems may interact with each other at their own will, important questions about such interactions are raised. *Who is responsible for verifying the correctness of adapted behavior coming from other systems? How do multiple behavioral adaptations take place whenever the situations leading up to them are present in the surrounding execution environment?* Even though the programming technology to enable dynamic adaptations of software systems with respect to their surrounding environment already exists, it still lags with respect to the support offered to address the aforementioned questions. Furthermore, two different trends are seen in the development of dynamically adaptive software systems. One trend focuses on offering novel and efficient techniques for dynamically changing the behavior of a system [159, 160]. The

second trend focuses on the definition, representation, and propagation of situations in the surrounding execution environment to which the system should adapt [158, 175]. Currently there is a mismatch between the technological advancement in these two trends.

The purpose of this dissertation is to propose a formal basis for dynamically adaptive software systems that can be realized in a single programming model thus, addressing the existing shortcomings in the technology for developing dynamically adaptive software systems. In particular, we propose a new programming model focused on the interaction between behavioral adaptations, facilitating the development of dynamically adaptive software systems by preventing inconsistencies in their behavior. We introduce **context Petri nets**, a formal theory and execution model for Dynamically Adaptive Software Systems (DASS) based on the low-level formalism of Petri nets [137], which allows to bridge the gap between system design and development. Moreover, having such a formal foundation makes it possible to use the model as a reasoning engine for the analysis and verification of behavioral adaptations and their interactions.

1.2 Problem Statement

The paradigm of Context-Oriented Programming currently offers one of the most dynamical approaches to adapt the behavior of software systems (see Section 3.1 for support of this statement). As consequence, these systems are most prone to behavioral inconsistencies. Therefore, we focus on such systems as our main object of study. In order to support the dynamic adaptation of a system's behavior according to the surrounding execution environment, COP languages are currently characterized by allowing us: the definition of semantically relevant situations in the surrounding execution environment, the association of behavioral adaptations with such situations, and the scope in which each behavioral adaptation has to take place. The term **adaptation** is hereafter used to refer to the dynamically introduced behavior presenting the aforementioned set of characteristics. Existing COP languages have also explored aspects of distribution [185, 163], response and propagation of changes in the surrounding execution environment [103, 9], and definition of interaction rules between behavioral adaptations [77], the principal concern of COP languages has been to prove the means to effectively modify a system's behavior at run time. Little attention has been paid to correctness or consistency between behavioral adaptations.

One of the most important requirements for dynamically adaptive software systems, mostly overlooked by existing COP languages, is to be able to ensure that the observed behavior of the system takes place as predicted during its conception, even in the presence of behavioral adaptations. That is, it must be ensured that all changes in the surrounding execution environment of the system propagate into a consistent composition of the system and its adapted behavior. As systems grow, the complexity of keeping track of the defined behavioral adaptations, and managing the scope and situations in which they are

applied increases, potentially leading to unpredicted behavior or **behavioral inconsistencies**. Behavioral inconsistencies are due to three main reasons [33]. First, adaptations are multi-dimensional. That is, different behavioral adaptations may be associated to the same situation of the surrounding execution environment, and a particular behavior (system functionality) may be adapted differently in different situations of the surrounding execution environment. Second, adaptations suffer from accidental interaction. There is little support provided during the software development process for the definition of allowed and disallowed interactions between behavioral adaptations. Interaction between adaptations can only de defined at a high-abstraction level, causing a mismatch between the intended interactions of adaptations and their implementation, which can lead to an accidental interaction between adaptations. Moreover, there is little support for verifying the interaction between adaptations. Third, adaptations are volatile. Behavioral adaptations may surface within the system unexpectedly at run time —that is, it cannot be predicted when the conditions of the surrounding execution environment will trigger a behavioral adaptation. Hence it is unfeasible to foresee all possible cases of interaction between behavioral adaptations.

The programming facilities currently provided by the COP paradigm still lack a programming model that allows to soundly analyze, prevent and manage behavioral inconsistencies. This increases the difficulty of developing such Dynamically Adaptive Software Systems. The challenges programmers of such systems are faced with include:

- Awareness of the correspondence and coherence between adaptations — that is behavioral adaptations, and the situations to which these apply.
- Awareness of all defined situations for which the system presents behavioral adaptations, and the run-time interaction between them —that is, what is the expected behavior of the system when behavioral adaptations are continuously being introduced to or withdrawn from the system.
- Manual exploration and simulation of *all* possible situations in which behavioral adaptations may be introduced to or withdrawn from the system.

Obviously, the presence of behavioral inconsistencies hinders the development of Dynamically Adaptive Software Systems. On the one hand, they encumber the usability and user acceptance of such systems. On the other hand, they impose difficulties on the conception and construction of robust and well-behaving Dynamically Adaptive Software Systems

1.3 Research Goals

This dissertation takes the definition of behavioral adaptations and the management of their interaction as a platform for the definition of a sound programming model for Dynamically Adaptive Software Systems. With this goal

in mind, we investigate different aspects of the software development process of Dynamically Adaptive Software Systems. More specifically, we provide the necessary formalization, investigate how this affects the programming language and run-time execution, and tool support. Based on these aspects, the principal research question addressed throughout the dissertation is:

How to ensure the consistency and predictability of dynamically adaptive software systems, in the presence of multiple behavioral adaptations, continuously being introduced to and withdrawn from the system?

The subsequent subgoals follow naturally from this initial research question:

G.1 Lack of interaction definition. The development of our work is to provide a means to express the type of interactions that adaptations should or should not exhibit between each other at run time. Specifying such interactions is important to ensure that the behavior of the system is always the most appropriate for its surrounding execution environment. Adaptations are not isolated at run time. Adaptations interact with each other, and their associated behavior may be influenced by the behavior associated to other adaptations. However, behavioral adaptations are usually developed independently, they can be defined at different times, or even by different programmers. A concise model expressing allowed and disallowed interactions between adaptations is needed.

G.2 Accidental interaction of adaptations. The development of our work is to allow the definition of interactions between adaptations in such a way that accidental interactions are avoided. Interactions between adaptations may be accidental by two main reasons. On the one hand interactions might be missed by not specifying them explicitly. On the other hand interactions may exist as adaptations become simultaneously available, if this was not foreseen. Defining all possible interaction rules between all adaptations can become cumbersome and error prone as systems grow.

G.3 Lack of verification. The development of our work is to integrate the verification of defined interactions between adaptations. Currently, there is little support provided for the verification of defined interactions between adaptations, diminishing the reliability on Dynamically Adaptive Software Systems.

G.4 Lack of property analysis. The development of our work is to enable the analysis of system properties for the introduction and withdrawal of adaptations. In order to allow for a lightweight verification process at run time, a complementing design/compile time analysis process is required. Such, a design-time analysis of the system is used for the early identification of errors, thus reducing the number of possible inconsistencies at run time. Moreover, a design-time analysis of the system could relief part of the run-time verification, which might be too heavyweight for the

small-powered devices that Dynamically Adaptive Software Systems are envisioned for.

G.5 Lack of a comprehensive programming model. The development of our work is to provide a comprehensive programming model for the development of Dynamically Adaptive Software Systems. Development of such systems currently lacks the means to define of adaptations, their behavior, and the situations of the surrounding execution environment in which they take place, rendering the development of such systems more complex.

1.4 Approach

Current COP languages suffer from a mismatch between the definition of behavioral adaptations and the situations in which they are applicable. Normally in current COP languages, behavioral adaptations are typically described within the programming language, while the situations in which behavioral adaptations should occur are defined using external frameworks, if defined at all. Additionally, interactions between behavioral adaptations are often specified through a high-level description of rules describing the intended interaction. Unfortunately, it is not possible to verify the correctness of such high-level definitions often presenting a mismatch between high-level specifications and the run-time representation. This creates inconsistencies between the expected and observed behavior of adaptation interactions.

In order to provide a sound programming model for the development of Dynamically Adaptive Software Systems, we begin by providing support for the interaction between behavioral adaptations. Such interactions must be defined formally and as close as possible to their run-time representation, in order to ease their verification and avoid the mismatch between formalization and implementation. Having a formal definition of the interaction between behavioral adaptations implies providing a formal definition of the adaptations themselves. Such a formalization is beneficial for three reasons. First of all, formally defining the interaction between behavioral adaptations allows reasoning about system properties and their correctness. Additionally, the formal definition could be used to capture any kind of accidental interaction that programmers may not have foreseen, facilitating their work and protecting the system from potential behavioral inconsistencies. Secondly, a formal definition of behavioral adaptations and their interactions that remains close to their run-time representation would eliminate the existing mismatch between the two. Such a formalization could provide a uniform foundation for different COP languages. Finally, since adaptations are already formally defined, the definition of the situations in which they are applicable can be included in the same formalism, instead of using an external framework. We argue that building a formalism with these characteristics, which can also be used as run-time representation of the system and its adaptations, suffices to have a sound programming model for Dynami-

cally Adaptive Software Systems.

The formalism allows to define adaptations, their interactions, and their behavior closely to the run-time representation of the system. Using the formal foundation as a run-time model of the system already covers most of the requirements for a sound programming model. The remaining aspect to be explored is then the definition of the situations of the surrounding execution environment in which adaptation are to take place, providing a sound and uniform basis that fosters Context-Oriented Programming systems.

In our exploration of a formal foundation for Dynamically Adaptive Software Systems we opted for a formalism based on Petri nets, which provides a formal description of the behavior of software systems, and is close to the their execution. The intrinsic operational and formal specification of Petri nets allows us to straightforwardly model, execute and reason about software systems. Other formal specifications, like Boolean logic or automata require to be combined with additional models or external tools to cover all three aspects. In addition, the Petri net model provides different extensions that can be used to support the dynamic and reactive characteristics of Dynamically Adaptive Software Systems; extensions that would otherwise have to be implemented using other formal specification. As we will later show in Section 3.2, the use of Petri nets for the development of our programming basis is motivated because, different from the other surveyed approaches, Petri nets (and their extensions) satisfy the requirements for a conflict resolution model and provide additional support for designing Dynamically Adaptive Software Systems. Moreover, we argue in favor of the appropriateness of Petri nets because the definition, execution and analysis of the system all remain within the same formal domain, easing the adoption and extension of such formalization as a basis of Context-Oriented Programming.

Our approach consists of shifting the concepts of Petri nets to the setting of Dynamically Adaptive Software Systems and using the Petri net model as a formal model for defining and executing behavioral adaptations. As a result, we are able to profit from the existing machinery of Petri nets (formal proofs and analysis tools) for the development, analysis, execution, and simulation of dynamically adaptive software systems.

To validate the appropriateness and effectiveness of the model we couple it to the Subjective-C COP language [33]. Our variation of the programming language is effectively used to develop applications that change dynamically according to their surrounding execution environment.

1.5 Contributions

This section highlights the main contributions of this dissertation.

Formalizing COP systems

Up until now, little work has been done in the formalization of COP languages. A formal semantics has only been defined for the execution and composition of behavioral adaptations [74, 38, 165, 2, 96, 105]. This dissertation develops a more comprehensive formalization that effectively comprises several other aspects of COP systems. That is, the definition of adaptations, the composition of their associated behavior, the context dependency relations defining the interaction between adaptations, and the activation of adaptations —that is, (de)composing behavioral adaptations with the system.

Run-time verification of inconsistencies

To deal with behavioral inconsistencies, we introduce a language-integrated model that allows defining interaction rules between adaptations. The specification of **context dependency relations** unequivocally defines the way adaptations interact with one another. Even more, context dependency relations are defined as transitive relations between adaptations. The transitive properties of context dependency relations is beneficial because it does not require programmers to foresee all possible interactions between adaptations, but only consider a more limited set of directly related adaptations.

Context dependency relations are used at run time to keep track of adaptations and their state, in order to ensure that there are no inconsistencies with respect to the defined context dependency relations during the system's execution. At run time, whenever the state of an adaptation is requested to change, the system automatically verifies that all constraints imposed by all context dependency relations are satisfied. If this is not the case, the system disallows the state change for the adaptation.

Identification of inconsistencies

Currently, there are no means to reason about COP systems, their properties, or their correctness. We propose an analysis engine to reason about system properties, in particular, about coherence and correctness of adaptations with respect to their context dependency relations. At design time, our programming model offers the possibility to analyze certain properties of the defined context dependency relations. The objective of such analysis, is to detect adaptations that cannot occur in the system, due to the constraints imposed by the context dependency relations.

Comprehensive programming model

Alongside the basis for developing Dynamically Adaptive Software Systems presented in the dissertation, accompanying tool support is introduced to facilitate the development of dynamically adaptive software systems. The most important technical contribution of the developed tools, lies in the provisioning of

a simulation environment in which programmers can test the interaction between adaptations without requiring a full-fletched application. Additionally, we facilitate the development process of COP systems by allowing the definition and introduction of adaptations and context dependency relations dynamically at run time. These features are not presented in any other COP programming model.

The formal basis for the development of Dynamically Adaptive Software Systems presented throughout these contributions is reified in our proposed programming model for COP systems, called context Petri nets. Context Petri nets enable programmers to easily design and develop different aspects of COP systems, such as the definition of adaptations, situations the system should adapt to, interaction between behavioral adaptations, and the system's reaction to changing situations from the surrounding environment; all while ensuring that the system is free of inconsistencies. Context Petri nets are the missing link, represented in Figure 1.1, between the formalization, execution, and analysis required to have a sound programming model for dynamically adaptive software systems, and Context-Oriented Programming in particular.

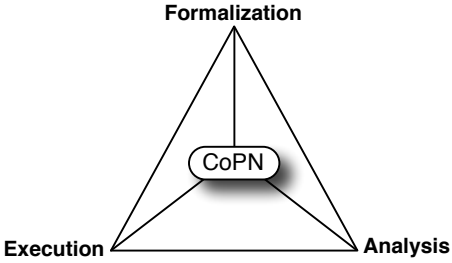


Figure 1.1: Positioning CoPN with respect to the programming model requirements of dynamically adaptive software systems.

1.6 Supporting Publications

The following (co-)authored publications support the key ideas in this dissertation:

- **Context Petri Nets: Enabling Consistent Composition of Context-Dependent Behavior** [33]
  Nicolás Cardozo, Jorge Vallejos, Sebastián González, Kim Mens, and Theo D'Hondt
  6th International Workshop on Petri Nets and Software Engineering (PNSE 2012)
  This paper proposes our Petri net based programming model for COP languages for the management of interactions between adaptations, detailed in Section 6.3 . The paper also presents the language API provided

to developers for the use of context Petri nets (detailed in Section 6.4). Additionally, this paper presents the maps application described in Section 2.3.3.

- **Uniting Global and Local Context Behavior with Context Petri Nets** [32]
  Nicolás Cardozo, Sebastián González and Kim Mens
  4th International Workshop on Context-Oriented Programming (COP 2012)
  This papers presents different scoping techniques used in COP, and unifies them in the programming model of context Petri nets with the introduction of colored tokens. This interaction is presented as part of our validation in Section 9.2.

- **Modeling and Analyzing Self-Adaptive Systems with Context Petri Nets** [30]
  Nicolás Cardozo, Sebastián González, Kim Mens, Ragnhild Van Der Straeten and Theo D'Hondt
  7th International Symposium on Theoretical Aspects of Software Engineering (TASE 2013)
  This paper presents the formalization of context Petri nets which constitutes the core of Chapter 6. The paper also presents the main ideas behind the design-time analysis implemented in context Petri nets, as presented in detail in Chapter 7. Additionally, the paper presents the mobile city guide case study used as part of our validation in Section 9.1.

- **Context-Oriented Programming for customizable SaaS Applications** [184]
  Eddy Truyen, Nicolás Cardozo, Stefan Walraven, Jorge Vallejos, Engineer Bainomugisha, Sebastian Günther, Theo D'Hondt and Wouter Joosen
  27th Symposium on Applied Computing (SAC 2012)
  This paper presents a comparison between the adaptability provided by COP and that provided by the dependency injection design pattern, which is used as a basis of the discussion presented in Section 10.2. Additionally, the paper introduces the web booking application described in Section 2.3.2.

- **Subjective-C: Bringing Context to Mobile Platform Programming** [77]
  Sebastián González, Nicolás Cardozo, Kim Mens, Alfredo Cádiz, Jean-Christophe Libbrecht and Julien Goffaux
  3rd International Conference on Software Language Engineering (SLE 2010)
  This paper presents Subjective-C, the COP language we use in this dissertation as a platform to prove the ideas of context Petri nets. The paper also presents our initial informal definition of context dependency relations. Finally, this paper presents the home automation application described in Section 2.3.1

1.7 Roadmap

The main contribution of this dissertation is the proposal of a theory for the development of systems that can dynamically adapt to their surrounding execution environment. Bellow we summarize the chapters of this dissertation.

Chapter 2: Requirements for consistent DASS provides an overview of the theoretical context in which this dissertation is developed. This chapter is used to describe the general characteristics of the kinds of systems we are interested in, and the situations that can give rise to behavioral inconsistencies. The chapters presents an overview of the characteristics of the inconsistency management process. We provide a set of examples of applications that adapt dynamically, motivating the requirements for highly dynamic adaptive software systems, and an execution model to manage inconsistencies in such systems.

Chapter 3: State-of-the-art in DASS and inconsistency management models presents the state-of-the-art in existing systems that enable dynamic adaptations of their behavior. This chapter also presents existing conflict resolution models that can be used for the run-time management of inconsistencies in Dynamically Adaptive Software Systems.

Chapter 4: Context-oriented programming provides an in-depth look into a particular class of Dynamically Adaptive Software Systems that fulfill all requirements listed in Chapter 2, namely Context-Oriented Programming (COP). We survey all existing COP languages, taking into account their main characteristics and the support they provide for managing inconsistencies. We then continue by presenting our laboratory language, Subjective-C, and its main characteristics.

Chapter 5: Petri nets provides an in-depth look into Petri nets, a particular formal model that realizes the requirements listed in Chapter 2 for conflict resolution models. We present existing analysis techniques used to reason about the properties of systems modeled with Petri nets. Additionally, we describe different extensions of the basic Petri net model that are later used in the definition of context Petri net.

Chapter 6: Modeling and managing DASS introduces context Petri net, giving the basics of its formal definition and the mapping between COP and Petri nets concepts. This chapter also describes the formal definition of context dependency relations and how these are used to generate complete COP applications. Moreover, we also describe the run-time semantics used to verify the system's consistency. This chapter is concluded by presenting the programming interface for using the model from within the Subjective-C COP language.

Chapter 7: Analysis and verification of DASS describes the analysis engine that comes with our context Petri net model for reasoning about the correct-

ness and coherence of adaptations and their associated behavior. This chapter describes the Petri nets analysis techniques available for context Petri net.

Chapter 8: A comprehensive programming model for DASS describes the programming model that comes with context Petri net. This chapter presents the details of the different components of the context Petri net programming model, and how these comply with the development process of Dynamically Adaptive Software Systems. Finally, the supporting tools developed with context Petri net for the acquisition of contexts and the simulation of context activations are presented in this chapter.

Chapter 9: Context Petri nets at work evaluates the theory developed with context Petri net in three main axes. First, we evaluate the usefulness of context Petri net as a reasoning engine for existing COP applications. Second, we evaluate the appropriateness of context Petri net as a theory for COP systems, by extending the COP model within context Petri net. Third, we present the extensibility of the model itself, by providing two new context dependency relations. Additionally, through tailored benchmarks, we provide a discussion about the efficiency of context Petri net at run time.

Chapter 10: Putting context Petri nets in perspective broadens the formal basis of context Petri net again by putting it in perspective with the larger context of Dynamically Adaptive Software Systems. This chapter thus opens the context Petri net model to Dynamically Adaptive Software Systems and discusses its appropriateness and usefulness for such domains.

Chapter 11: Conclusions wraps up the dissertation by restating the contributions in a fine-grained and more technical manner. The chapter discusses the current limitations of our work and then suggests other avenues in which research in context Petri net could be further developed.

Chapter 2

Requirements for Consistent Dynamically Adaptive Software Systems

This dissertation focusses on managing inconsistencies in Dynamically Adaptive Software Systems. Hence, we divide our work in two main fronts. On the one hand, we are interested in software systems that can adapt their behavior dynamically according to the changing conditions of their surrounding execution environment. This is motivated by the observation that nowadays software systems have nearly unlimited access to an immense variety of information. To benefit from this information, software systems may be customized or adapted dynamically providing a richer experience and usability to their users, even more so when adaptations take place as the system runs. On the other hand, to increase robustness of such systems, we are interested in identifying and managing behavioral inconsistencies that may occur due to adaptations taking place at run time.

The development of software systems that adapt dynamically their behavior according to the situations of their surrounding execution environment is not without challenge. One particular concern that becomes apparent in such a setting is the *predictability of the system behavior*. It is particularly difficult to know when a system is supposed or allowed to adapt to particular information received from its environment, since information is received unannounced and it is very hard to anticipate all possible changes before hand. Current software development technology is not well-equipped to deal with such situations. A process to manage the behavior of the system in the presence of run-time adaptations is required to preserve its predictability.

This chapter continues in Section 2.1 by providing a definition of Dynamically Adaptive Software Systems and describing different types of dynamicity and adaptability. Section 2.2 presents the inconsistency management process. Section 2.3 presents different scenarios that illustrate situations in which inconsistencies may be yield by the system. Such situations put in evidence the symptoms of behavioral inconsistencies, motivating the requirements for consistent Dynamically Adaptive Software Systems described in Section 2.4.

2.1 Dynamically Adaptive Software Systems

Long-lived software systems need to take software evolution into account [122], since such software systems are being adapted in order to cope with their changing requirements; even more so when software evolution occurs continuously at run time. Some of the reasons why the requirements of software systems may change during their execution are the need for introducing and removing functionality, optimizing algorithms, or customizing the system for particular users. In the following we precise the notions of dynamic adaptation, inconsistencies, and the situations in which unpredicted behavior can arise, as a frame of reference for the rest of this dissertation.

An **adaptation** is referred to as a system modification that has an intrinsic impact on its behavior. Adaptations can take place at different moments in the life cycle of a software system such as design, development, or maintenance. In this dissertation we are interested in **dynamic adaptations** —that is, adaptations that occur while the system is executing. Note also that our notion of dynamicity includes the interaction of the system with its surrounding execution environment.

Definition 2.1. ***Dynamically Adaptive Software Systems*** *are defined as software systems that can automatically trigger pre-defined behavioral adaptations of the system,*[1] *as a result of information acquired via, for example, sensor networks or system monitors.*

Each adaptation defined on the system concerns one specific situation of its surrounding execution environment. An adaptation may cover different units of functionality, or different fragments of behavior (e.g., methods or functions). An example adaptation is the rotation of a mobile device from portrait to landscape and *vice versa*. In such a case, the situation to adapt to is the change in the orientation of the device (information gathered throughout an integrated motion sensor), and the behavioral adaptations associated with this situation are, a modification in the displayed information and an update of the functions drawing of the device's layout. Nonetheless, when adaptations occur unannounced over time, unforeseen behavior of the system may arise. For example, if a behavioral adaptation is removed, and this was supposed to be used by an adaptation already composed with the base system. In the device orientation example this could occur if the redrawing adaptation is removed, but the adaptation modifying the information is not. An example of an undesired interaction between adaptations is that of two different adaptations modifying the same functionality. These can occur at the same time in the system when the specific situation defining them are signaled by the sensor network as simultaneously present in the surrounding execution environment. We distinguish between various situations in which inconsistencies may arise:

[1] Pre-defined behavior in this context means that the adaptation is not inferred, for example by means of artificial intelligence techniques, but rather that the behavior has been defined by programmers but is not yet composed into the system.

- If there is no defined order in which behavioral adaptations should take place, any of the available adaptations could be applied, possibly providing a different behavior for different instances of the system. The predictability of the behavior is compromised in this situation.

- Even when the order in which behavioral adaptations execute is defined, and assuming they are all executed (e.g., executing one after the other), the behavior provided by the adaptations may be contradictory. That is, one adaptation may undo the actions performed or expected by another. Such behavior is not desired.

- The dynamic introduction and withdrawal of behavioral adaptations may affect the consistency of the system's behavior. Withdrawing a behavioral adaptation from the system can cause a behavioral inconsistency if removed behavioral adaptations are expected to be used by other adaptations currently executing. For example, let us order behavioral adaptations based on the functionality increment each adaptation provides to the basic behavior, where the behavioral adaptations that provides the smallest functionality increment to the basic behavior is executed last. If during the execution of the system, one of the adaptations is removed dynamically, a mismatch could exist between the expected system state the succeeding behavioral adaptation expects and the state provided after the execution of the preceding behavioral adaptation. The behavior of the system can then lead to erroneous or undesired states.

  Introducing a behavioral adaptation into the system can cause an inconsistency of its behavior if the introduced behavioral adaptation interposes the current execution order of other behavioral adaptations already composed into the system. For example, the inserted behavioral adaptation could modify the state of the system in a way not expect by succeeding behavioral adaptations, causing an inconsistent or erroneous system state when the later behavioral adaptation executes.

These examples of interactions between adaptations are regarded as inconsistencies because it is either not possible to predict the behavior of the application when the adaptations are combined, or because the observed behavior is not the expected one.

Definition 2.2. ***Inconsistencies*** *are defined as situations in which the application shows unpredicted behavior (functionality with contradictory or erroneous behavior). Inconsistencies may not necessarily break or crash the system, but rather only show unexpected behavior.*

Clearly, inconsistencies need to be dealt with carefully. The following section presents the process of dealing with inconsistencies.

2.2 Inconsistency Management Process

To deal with inconsistencies of Dynamically Adaptive Software Systems, as presented in the previous section, we take inspiration from the inconsistency management process defined for other domains such as Model-Driven Engineering (MDE) [132, 189] or requirements engineering. In this section we define exactly what we mean by *identifying* and *managing* inconsistencies.

The problem of managing inconsistencies between software artifacts has been widely addressed in the modeling community. In the context of MDE, in particular, a vast body of research has been conducted around the topic of managing inconsistencies between different model artifacts (e.g., activity diagrams, sequence diagrams, and other UML diagrams) describing a software system. The process proposed for the management of inconsistencies in software engineering [176] consists of:

- A *detection of inconsistencies* activity which systematically checks the different model artifacts of a software system, for example, for overlapping descriptions of the system, or violations of predefined consistency rules for a particular model. Detection of inconsistencies is typically done by means of logic-based rule semantics, model checking, or specialized automated analysis.

- A *diagnosis of inconsistencies* activity which identifies the source, cause, and impact of an inconsistency. Inconsistency diagnosis is related to the inconsistency detection step in the sense that, in order to diagnose the presence of an inconsistency, it needs to take into account the predefined set of consistency rules that models need to comply with. Diagnosis of inconsistencies is usually performed by generating an abstraction of the system —that is, an abstract model covering all concrete model artifacts. Such models are then used to automatically verify some consistency rules defined about the system. If the verification fails, the source of the inconsistency should be revealed; which is a challenging problem on its own.

- A *handling of inconsistencies* activity which evaluates the various corrective actions that could be taken to solve an inconsistency once it is identified and diagnosed. The process of handling inconsistencies usually takes into account the evaluation of the costs, benefits, or risks of applying a corrective action. Inconsistency handling can be performed at run time or at earlier stages of the software development process.

In this dissertation we do not undertake the complete process of inconsistency management as presented above, but focus only on the *detection* and *handling* of inconsistencies activities. The *diagnosis* activity will be the focus of further research.

Definition 2.3. *Herein* ***inconsistency management*** *is referred to as the activity of avoiding situations in which inconsistencies can arise, and the activity of verifying satisfiability of interaction rules between adaptations at run time.*

Inconsistency management in Dynamically Adaptive Software Systems is achieved by means of the definition of a semantically sound set of rules for the interaction between software adaptations, and a programming model that enables the run-time verification of said rules.

2.3 Motivating Examples

This section presents example situations in which software systems could benefit from the dynamic adaptation of their behavior. The different examples are used as witnesses of behavioral inconsistencies yield in the system. The examples presented on this section serve as motivation for the definition of the requirements for a consistent Dynamically Adaptive Software Systems presented later in Section 2.4, and are also used throughout this dissertation to put different aspects of Dynamically Adaptive Software Systems into perspective.

2.3.1 Home Automation

Home automation systems are common examples used in the setting of Dynamically Adaptive Software Systems [34, 77]. Home automation systems allow to regulate different household appliances and services such as a stereo or a tv, or room temperature or lighting. The services offered by the room can be customized according to the room equipment (e.g., configuration of windows) or the user currently in the room (e.g., through user preferences about lightning and temperature).

We take as a concrete example the user detection system in the setting of a home automation environment. Throughout the house a sensor network is deployed to detect users, for example, by the identity of their mobile devices or specific RFID tags. The house appliances are adapted to the user requirements and specifications. For example, use the room speakers to play the user's playlist, call the authorities in case of burglary, or inform the authorities in case of an emergency in the house (e.g., fire or flood), or medical emergencies for one of the hose inhabitants.

Using the home automation scenario and its desired adaptation for the user detection concrete example, it is possible to identified the desired requirements for the application.

Reaction to changes The application should react to changes in its surrounding execution environment in order to adapt its behavior accordingly. As users enter a room, the behavior of the services and appliances deployed in the room should adapt to the user preferences.

Customization Home environments are not static environments. Rather they evolve over time, for example by introducing new services, new appliances, or new users. Customizations with respect to such changes to the environment should be able to be introduced with ease. For example, introducing an air conditioning service for all rooms in the house requires

other services (e.g., the heating service) already deployed in the application to adapt to the behavior of the air conditioning service. Moreover, the air conditioning service needs to adapt to the user preferences.

Transience Adaptations can be introduced at any moment during the system's execution. As a consequence adaptations should be prepared to interact with other adaptations as they are introduced, as well as with the base behavior of the application. Moreover, adaptations can not only be introduced, but it is also possible to remove them from the application. This means adaptations are only available for a defined period of time, for example, the time a user is in a room.

2.3.2 Web Booking

The web booking application is a highly configurable service that travel agencies can use for booking hotels on behalf of their customers [184]. Employees of the travel agency are offered a customized user interface and customers of the travel agency can login to check the status of the travel items through a URL with a custom-made domain-name that corresponds with the travel agency. A special administrator role is assigned to someone who is responsible for configuring the application, setting up the application data and monitoring the overall service, for example an staff member of the travel agency.

Take the case of a particular travel agency that wants to be able to offer discounts to their returning customers or during the low season. The web booking application should be extended with an additional service for managing customer profiles and an adaptation on the service for calculating booking prices. Let us assume furthermore that the base application is offered to customers at no or low cost, but travel agencies incur an additional price for additional services. Based on this simple customization scenario, we can derive requirements with respect to application development, configuration, adaptation, and run-time support.

Isolated software adaptations The application should be offered a simple way to manage the different travel agency-specific adaptations as separate units of deployment that can be selectively bound to the core architecture of the application.

Configuration facility With respect to customization, travel agency administrators should be offered a configuration facility to select what software variations should be enabled for them (e.g., the price calculation service). In addition, this facility should also allow to specify specific configuration parameters (e.g., business rules for the price calculation service). These configuration data should be isolated within the application from the information of other travel agencies.

Run-time activation of adaptations Run-time support is needed to provide support for activating behavioral adaptations for each of the travel agencies or for each of their customers. When a user (either customer or

employee) logs in, the travel agency to which the user belongs should be determined. Based on the acquired travel agency ID, the run-time support should then activate the appropriate behavioral adaptations to process the requests of the user. Another key requirement of the run-time support is that the agency-specific behavioral adaptations should be applied in an isolated way without affecting the service behavior that is delivered to other travel agencies or users of that agency.

2.3.3 Context-Aware Maps

The context-aware maps example is small enough to be easily grasped, yet complete enough to capture the intuition of the dynamic aspects of behavioral adaptations and their interactions [33].

The context-aware maps application consists of a basic map visualization service decorated with information about different places, public transportation stops and buildings. By adapting the basic map system with a `Positioning` service context, the system can provide an enhanced map experience by taking into account the current geographical location of the user.

The general `Positioning` service context can be further refined by more specific positioning services. For example, a `GPSAntenna` service retrieves the current location using the device's GPS, a `GSMLocation` service calculates the current position with methods like the time difference of arrival (TDOA) using the GSM signal from cellular network cells, or near location based services (`NLBS`) which are used to calculate the current location indoors (e.g. buildings). Near location positioning services are defined for the context-aware maps application to represent each of the available connection services of the device, `WLAN`, `Infrared`, or `Bluetooth` to mention some examples.

The current geographical location of the user can be calculated by using one (or a collection) of the positioning services described above. How the actual method is chosen to calculate the position is unimportant to users. Positioning services are automatically chosen based on the gathered data through the sensor network the device is associated with. For example, the `WLAN` and `GSMLocation` methods can be used when the application is running on a mobile phone with access enabled for wireless connections.

The user is implicitly aware of the positioning service used through the application's graphical interface. The service to display the user's current location takes into account the service to calculate the user location by placing the location in a colored circled area. The area surrounding the user's location is displayed in function of the accuracy of the positioning service used.

The maps application also defines a `Private` adaptation to protect the user's information. User information, such as its location or identity, is undisclosed when this adaptation is active. The `Private` adaptation is useful, for example, when the user has an insecure connection. The `Private` adaptation provides the behavior to conceal all sensitive information regardless of the surrounding available services or incoming/outgoing communications. In particular, it

conceals the user's position by not providing the exact location (e.g., "at the bank") and by not displaying the identity of the user (i.e., the display image only shows a pin with no further information).

The fact that adaptations can be simultaneously active causes a number of difficulties for the design and implementation of applications that adapt dynamically at run time. Consider for example the following situations:

Adaptations interactions The NLBS location service provided by the application is not standalone. In order to offer the NLBS service, the system must first be able to offer a CONNECTIVITY service through, for example, a BLUETOOTH or WLAN connection. This condition needs to be checked every time the NLBS service is to become available. The system must track and coordinate *adaptation interactions*.

Multiple activations of behavioral adaptations Adaptations interactions can be used to define services that enable other services. This is the case for the POSITIONING adaptation, which can be made enabled every time one of the geographical location services becomes available. Conceptually, this means that if two geographical location services are available, the POSITIONING adaptation is enabled two times. We refer to the fact that adaptations can be activated several times as *multiple adaptation activations*, implementation of this capability is referred to in the literature as *activation counters* [74, 31]. The concept of multiple adaptations activations is introduced to enable interaction between adaptations, where the system must ensure that adaptations remain enabled as long as at least one of the services enabling it. For example, the POSITIONING adaptation should remain active as long as there is at least one geographical location service is available.

Conflicting behavior interaction Behavioral adaptations that provide contradictory behavior may be simultaneously available in the system. For example, the behavior associated to the PRIVATE adaptation conceals the user's location, while the behavior associated to the POSITIONING adaptation broadcasts the user's location. When the two adaptations are simultaneously available it is unclear if the user's position should be disclosed or not (i.e., which of the two behavioral adaptations should be used). This kind of *conflicting behavior interaction* must be avoided.

2.4 Requirements for the Consistency of Dynamically Adaptive Software Systems

This section describes the requirements for a programming model that allows managing inconsistencies in the context of Dynamically Adaptive Software Systems. The requirements are defined around the initial research goals described

in the problem statement (Section 1.3). *How to ensure the consistency and predictability of the system behavior, in the presence of multiple behavioral adaptations, continuously being introduced to and withdrawn from the system?*

To answer this question, we divide the requirements into two sets. First, we define the requirements for highly dynamic adaptive software systems (**D.**) supported by the requirements found in the different application scenarios described in Section 2.3, and using requirements previously defined for different Dynamically Adaptive Software Systems such as, dynamic software upgrades [114, 87], and self-adaptive systems [159].

D.1 *Timeliness*: Dynamically Adaptive Software Systems must allow adaptations to occur at any moment in time. This is due to the fact that adaptations occur as a result of the information gathered about the surrounding execution environment of the system. Since the system, has no control over gathered information in the general case it is not possible to predict when adaptations must be applied, neither whether an adaptation is safe (i.e., its composition with the system does not yield inconsistent states). Nonetheless, given that adaptations correspond to situations in the surrounding execution environment, we require Dynamically Adaptive Software Systems to reflect these situations by applying adaptations as promptly as possible. This requirement is motivated by the reaction to changes of Section 2.3.1 and the timeliness property of software upgrade systems [114].

D.2 *Granularity*: Dynamically Adaptive Software Systems must offer the possibility to adapt all entities of the system, from the most atomic operations, to the most coarse components and everything in between. Adaptations may vary in their level of specificity. On the one hand, one adaptation may completely replace a basic functionality provided by the system. In such a case, the corresponding adaptation would replace the whole component in charge of such functionality. On the other hand, an adaptation may only require a small increment of an already existing functionality. In such a case, the adaptation should only modify this functionality. This requirement is motivated by the configuration facility of Section 2.3.2 and the flexibility property of software upgrade systems [87].

D.3 *Independence*: Adaptations should normally be defined independently from the basic behavior provided by the system. Adaptations could be seen as overwriting or complementing the behavior already provided by the system. Definition of adaptations must be cleanly separated from the base logic of the system and isolated from other adaptations. This requirement is motivated by the isolated software adaptations of Section 2.3.2 and the conflicting behavior interaction of Section 2.3.3.

D.4 *Compatibility*: Adaptations react to situations in their surrounding execution environment. This allows us to define "temporal" adaptations —that is, adaptations that do not persist through the whole life of the system,

but that are only observable during a limited period of time. Moreover, during the time an adaptation is applicable, other adaptations may also be so. Dynamically Adaptive Software Systems must ensure that all adaptations are compatible with each other, as they are introduced to and withdrawn from the system. This requirement is motivated by the the independence of adaptations one in Section 2.3.1, adaptations interactions of Section 2.3.3, and the what question of self-adaptive systems [159].

D.5 *Extensibility*: The definition of an adaptation does not necessarily have to be known beforehand by the system. New adaptations could be defined and introduced while the system is running. Dynamically Adaptive Software Systems must allow the introduction of new adaptations without conflicting with those already defined. This requirement is motivated by the customization of Section 2.3.1 and configuration facility of Section 2.3.2.

Having defined the requirements for Dynamically Adaptive Software Systems, we use the process for inconsistency management presented in Section 2.2 to define the requirements (**M.**) for **conflict resolution models** of such systems —that is, models used for the management of software systems and the identification of inconsistencies. This requirements are to be applied to the particular case of Dynamically Adaptive Software Systems as defined in Definition 2.3.

M.1 *Interaction*: Interaction is central to dynamic adaptability. If adaptations do not interact with each other, the system is but a set of independent pieces of consistent behavior. However, if interaction between adaptations exists and is unaccounted for, the system may yield inconsistent states. Dynamically Adaptive Software Systems require a model that expresses the interaction between their adaptations. However, models that manage interaction between adaptations usually suffer from a state explosion problem due to the fact that *all* possible interactions *must* be defined beforehand. Whenever possible, interactions between adaptations should be transitive, in the sense that not all interactions must be explicitly defined, reducing the state expansion problem.

M.2 *Safety*: Adaptation definitions may be unknown by the system at the moment of its deployment. Adding new adaptations that the system has not previously taken into account can be harmful. Newly introduced adaptations may break the behavior provided by already existing ones, for example, by providing a contradictory behavior. Whenever an adaptation is introduced into the system, it should be ensured that this does not lead to inconsistent system states. A model for Dynamically Adaptive Software Systems must provide the means to verify the safety of the system.

M.3 *Abstraction*: The abstraction property of a model refers to the capability to successfully represent (at run time) the system and its states (e.g., adaptations, relevant information from the surrounding execution environment, interactions between adaptations and so on) in a way that they

can be used for the verification of properties of the system. To support the identification and management of inconsistencies in Dynamically Adaptive Software Systems a good abstraction model must be able to concisely express the different states of the system and the transitions (actions) between such states.

M.4 *Decision*: The decision property of the model refers to the analysis capability of the model to determine the satisfaction of system properties. A model for Dynamically Adaptive Software Systems must be able to reason about the consistency of system behavior even in the presence of dynamic adaptations.

2.5 Conclusion

This chapter overviews the domain of Dynamically Adaptive Software Systems bringing forward its principal characteristics and the situations in which the dynamic adaptation of the system's behavior may yield inconsistencies. To address such inconsistencies we overview the process of inconsistency management.

This chapter sets the requirements for highly dynamic software systems. Requirements **D.1** through **D.5** are defined according to the neecesities discovered in the development of different case studies using dynamic adaptations (Section 2.3), and existing requirements in the literature of Dynamically Adaptive Software Systems. To address the situations that may yield inconsistencies in the execution of Dynamically Adaptive Software Systems, Requirements **M.1** through **M.4** are defined taking inspiration from the inconsistency management process.

A programming model for Dynamically Adaptive Software Systems that conforms to the requirements put forward in this chapter renders such system highly dynamic, while preserving a consistent behavior of the system in presence of adaptations.

Chapter 3

Dynamically Adaptive Software Systems and Models for Inconsistency Management

This dissertation addresses the problem of managing behavioral inconsistencies in DASS. For this purpose we develop a formal basis for the development software systems that allows a consistent and dynamic adaptation of their behavior with respect to their surrounding execution environment. The objective of the formal basis is to avoid unpredictable system behavior in presence of dynamic adaptations, and to aid programmers in identifying situations which could yield inconsistent behavior. This chapter explores the two domains in which our research takes place. Therefore, we divide the problem of managing behavioral inconsistencies in Dynamically Adaptive Software Systems into two main bodies of work: approaches that realize Dynamically Adaptive Software Systems, and approaches that propose models for conflict resolution and inconsistency management of software systems. Each of these domains comprise a vast body of work. We limit the scope of this background section by focusing on those approaches that are explicitly concerned with *consistent adaptation of behavior at run time*.

In what follows we first provide an overview of the design space of Dynamically Adaptive Software Systems. We explore state-of-the-art techniques that allow behavior adaptation at run time, ranging from high-level architectural techniques, to language-specific techniques. Each of the presented techniques is evaluated with respect to the requirements **D.1** through **D.5**. The objective is to identify Dynamically Adaptive Software Systems that are highly flexible and dynamic, the work developed in this dissertation concentrates in these kinds of systems. Second, we present an overview of the design space of conflict resolution, and behavior management models for software systems. State-of-the-art approaches are divided into three categories covering high-level abstractions and frameworks, formalization approaches, and models for the structural representation and dynamic execution of systems. Each of the presented approaches is evaluated with respect to requirements **M.1** through **M.4**. The objective is to identify the models that can effectively manage inconsistencies when the

behavior of software systems is adapted at run time.

For each of the surveyed approaches we discuss their limitations and strengths with respect to the development of highly dynamic software systems that adapt to their surrounding execution environment. The formal basis for the management of Dynamically Adaptive Software Systems established in this dissertation (cf. Chapters 6 and 7) addresses the problems found in the surveyed approaches. Chapter 10 evaluates our work with respect to the surveyed approaches.

This chapter is rounded off by relating the surveyed techniques and models for the realization of consistent behavioral adaptations at run time to the requirements described in Section 2.4.

3.1 Realizing Dynamically Adaptive Software Systems

This section explores both the state-of-the-art, and well established mechanisms used in software systems to enable the adaptation of their behavior dynamically. In this dissertation we aim to address highly dynamic software systems, where different system entities (e.g., variables, processes, components) can be adapted at any moment in time as a response to changing events from external (or internal) events in the execution environment of the system. We start with the observation that adaptations of the system can take place at any moment during its execution; adaptations respond to unannounced changes. Furthermore, an adaptation should be able to be combined with any other adaptation without compromising the expected functionality of the system. Moreover, it is desired for the system to be able to incorporate with ease new adaptations as it evolves over time. This section presents an overview of the design space of Dynamically Adaptive Software Systems.

To evaluate the design space of Dynamically Adaptive Software Systems, we divide it in three categories according to the abstraction level in which particular mechanisms enabling dynamic behavior adaptation are implemented. The categories are: architectural solutions, middleware solutions, and programming language solutions. For each of these categories we explore different techniques implementing dynamic adaptive behavior. For each technique we discuss the challenges or shortcomings of the mechanism with respect to the five main requirements for Dynamically Adaptive Software Systems: timeliness, granularity, independence, compatibility, and extensibility of adaptive behavior.

3.1.1 Architectural Solutions

The category of architectural solutions explores well established techniques for the implementation of software systems which offer structured and reusable solutions to enable adaptation of behavior at run time.

Design patterns

Design patterns are widely used to structure, modularize and define architectures of software systems [68, 25, 66]. Different patterns allow the adaptation of behavior at run time by requiring the definition of adaptations to be integrated in the structure of the system. For example, adaptations are introduced as part of the class hierarchy, in the case of object-oriented systems. Although design patterns allow behavioral adaptations of the system, systems implemented using these solutions are rigid. Adaptations are required to be foreseen and need to adhere to the structure of the system, which is not always possible. This makes the conception of behavioral adaptations through design patterns cumbersome, and difficult to maintain. Among the patterns used for enabling dynamic adaptation, the most prominent are: *state pattern*, *strategy pattern*, *decorator*, *abstract factories*, and *dynamic proxies*.

We do not discuss design patterns in detail here because their shortcomings in providing behavioral adaptations at run time have been addressed using language solutions [72, 161], which we discuss later in Section 3.1.3.

Dynamic software upgrades

Software upgrades are commonly performed for introducing new features, bug fixes, or patches to a software system. In order to provide an update, normally the system has to be stopped and restarted with the new functionality. For many software systems downtimes are critical or even unaccepable. Whenever this is the case, a new approach called dynamic software upgrades [78, 187] is used, addressing the need to upgrade the behavior of software systems without stopping them. Dynamic software upgrades ensure *correspondence* and *safety* between different program versions. Four requirements have been proposed to support these properties of dynamic software upgrades [114, 87, 152], namely *timeliness*, *robustness*, *flexibility*, and *practicality*, in addition to other non-functional requirements such as *platform independence*, *performance overhead*, or *non-intrusion of the program architecture*.

Remember that the requirements defined for Dynamically Adaptive Software Systems are inspired in some of the requirements for dynamic software upgrades. However, most approaches for dynamic software upgrades only allow forward evolution of adaptations —that is, adaptations can be introduced to but not removed from the system. This presents a mismatch with respect to the vision of Dynamically Adaptive Software Systems where adaptations are composed in and out of the system according to the surrounding execution environment. As a result the two sets of requirements are different.

Timeliness: Timeliness stands at the core of dynamic software upgrades. When an upgrade is taking place, the system itself should continue working normally. As far as possible, upgrades should take place immediately after they are requested.

Flexibility: System upgrades can take place for any element defined in the system at any time.

Robustness: When upgrading a system it needs to be verified that no program inconsistencies arise, for example, by accessing one of the elements being upgraded while the upgrade is taking place. Normally, to avoid inconsistencies software upgrades must wait until a quiescense state to be composed in the system [114].

Practicality: Definition and installation of software upgrades should be as transparent as possible for programmers —that is, without disrupting the architecture of the base system. Programmers must be oblivious to the process in which upgrades are installed. In addition, if it is not possible to perform the upgrade, programmers should be notified about the error and (if possible) its cause.

Different approaches implementing dynamic software upgrades exist ranging from specific virtual machines, Bytecode manipulation [101], or software transactional memory [145]. Most of the existing approaches concentrate only on a subset of the characteristics given here, thus presenting weaknesses in other ones [87]. We highlight the main difficulties when implementing a system for dynamic software upgrades. Since ensuring the consistency or safety conditions for any kind of upgrade is a challenging task, many systems restrict the kinds of upgrades that are allowed. For example, by disallowing the modification of the class hierarchy, or modification of objects that have already been used. Behavior adaptations must always be planned ahead in dynamic software upgrades. Additionally, dynamic software upgrades are usually conceived to go forward in time, thus not providing support to revert upgrades previously introduced. Even more, approaches that provide such support, can only revert the current upgrade to its prior version. Finally, most dynamic upgrade systems provide either fine or coarse granularity for behavioral adaptations, but not both. As there exist cases that could benefit from having both approaches, this is a shortcoming of current dynamic software upgrade approaches. For these reasons we claim that dynamic software upgrades are not ideal for the implementation of Dynamically Adaptive Software Systems.

Section 10.1 discusses how the formal basis for the development of Dynamically Adaptive Software Systems presented in this dissertation could be used in the setting of dynamic software upgrades.

Software product lines

Software Product Lines (SPLs) are a structured approach to product design and customization [35]. SPLs allow defining adaptations of a product with ease, by transforming components of the system at run time.

In SPLs, definition of adaptations (usually called variations in the literature) is included as part of the regular software development process. Adaptations are expressed in a *variability model* (usually resembling feature diagrams [125])

and can be applied to the base system by means of model transformations. The design and customization process of SPLs includes the definition and transformation of the system's models —that is, adaptations are defined at design time, and model transformations are pre-processed at compile time. However, more and more SPLs approaches are shifting to allow adaptation and generation of products at run time. These approaches are named dynamic SPLs [82]. In a dynamic SPLs component transformations take place at run time. There are different techniques enabling such transformations, for example, regenerative, composable components construction, or incremental-move [35]. These techniques can be characterized mainly by performing the model transformation in two steps, *synthesis* and *modification*. In the synthesis step, a new base model is generated using the new configuration that gathers applicable adaptations. Each of the adaptations indicates a particular module to be replaced. Each substitution is updated in the original base model. In the modification step, the differences between the original base model and the generated base model are calculated. These two steps are supported by an internal rule engine of the system, gathering all substitution rules, evaluating from the space of possible adaptations, which of them are feasible.

Figure 3.1 shows an example of the high level design of the home automation application described in Section 2.3.1 alongside its variability model for emergency and burglar situations. Each of the situations (triangles denoted as variation points (`vp`)) is represented in Figure 3.1 by an independent variability model with two adaptations (squares denoted as variations (`v`)). The model in Figure 3.1 takes inspiration from the home automation system presented by Cetina et al. [34] and the adaptation model of Hallsteinsen et al. [82]. The lines going from adaptations to the base model denote, for each adaptation, the point where it is applicable.

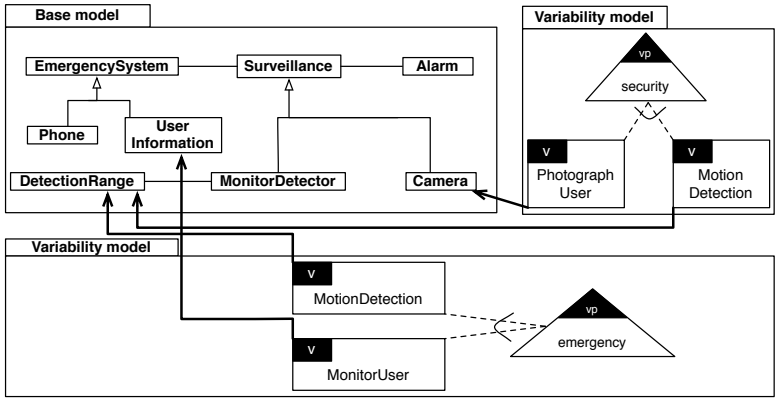


Figure 3.1: Base and variability models for the user detection system.

Using dynamic SPLs effectively allows the reconfiguration of systems at run time. However, system adaptations need to be foreseen in order to create the

variability model and specify the concrete transformation strategy. Moreover, the points of the program in which adaptations should take place must also be defined beforehand. Definition of variations and substitution rules can become cumbersome as complexity of the system increases. Interaction between adaptations becomes an issue when multiple substitutions can be applicable. The order in which substitutions take place needs to be manually encoded in the rule engine, where rules can be missed, or be too general not applying to particularly exceptional situations. Hence, SPLs are not ideal for the implementations of Dynamically Adaptive Software Systems.

Conclusion

Architectural solutions have as a main drawback the requirement of defining all adaptations beforehand. This may be unfeasible for the setting of highly changing environments. Additionally, adaptations are usually expressed in a coarse-grained or fine-grained fashion, but not both, restricting the kind of adaptations that can be defined by the system. Nonetheless, since adaptations are known beforehand, they can be introduced timely into the system. Similarly, architectural solutions are usually developed within the base system architecture, rendering these adaptations highly compatible between each other, and the base system.

3.1.2 Middleware Solutions

The category of middleware solutions explores platforms or systems focused on reducing the complexity of building software systems that can dynamically adapt their behavior, by shifting this complexity from the application design towards a reusable middleware architecture. Most of the existing middleware approaches that deal with adaptation of a system at run time are targeted to work in a distributed environment. That is, they take into account communication issues between different components of the system. Since communication between remote components is outside the scope of this dissertation, in this section we only discuss the aspects of the middleware related to dynamic adaptation of a system. Readers interested in the communication aspects of the middleware presented in this section, are encouraged to follow the references given to each of the approaches.

Dependency injection

Dependency injection, sometimes referred to in the literature as Inversion of Control [67], is a well-known design pattern for component-based applications that separates the management of component dependencies from the application base code.

Dependency injection increases component reusability by reducing the dependencies among them. Dependency injection is most commonly used in the

case of component composition, by enabling reusability of components. Subcomponents may freely change as long as they respect a common interface. Dependency injection is used as a middleware to dynamically adapt the behavior of a software system, by implementing it as a Platform-as-a-Service (PaaS) solution [191]. That is, platforms natively provide the means to inject behavior dynamically, for example, using dedicated language constructs. However, the inherited component-based composition mechanisms used in dependency injection approaches still present some limitations.

Figure 3.2 shows a schematic version of the web booking application presented in Section 2.3.2.

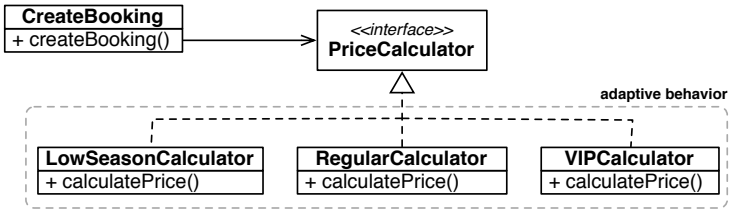


Figure 3.2: Dependency (interface) injection diagram for the web booking application.

There are three main techniques to enable dependency injection, namely *construct injection*, *setter injection*, or *interface injection*.[1] These three techniques require similar modifications to the structure of the system. Snippet 3.1 shows the case for interface injection as it is performed in Guice,[2] A lightweight dependency injection framework for Java 5 . First, objects need to include an instance variable to define the injected component, the `configuration` variable in Line 3. Second, provide a method through which the component can be injected into the instance variable (the class constructor, the variable setter, or the interface injection method, depending on the used injection technique), example injection methods are shown in Lines 16 and 21 where the `calculatePrice` method is defined. Third, the possible components to be injected in the instance variable need to be defined through a configuration step, shown by the method in Lines 11 through 14.

Dependency injection can effectively provide different behavior of a software system through the management and dynamic injection of application components. In the web booking application, this is done by the injection of different price calculation components based on specific conditions, for example, type of user, time of the year for which the booking is made, and so on. Different components are dynamically injected into the application according to such conditions.

Essentially, using dependency injection, systems are designed as SPLs with

[1] The interface injection technique is closely related to the *abstract factory* design pattern [68].

[2] `http://code.google.com/p/google-guice/`

```
1  //Calling the price calculation service
2  public void createBooking() {
3    if(configuration == LOW_SEASON) {
4      LowSeasonCalculator calculator = new LowSeasonCalculator();
5      Injector injector = Guice.createInjector(calculator);
6      PriceCalculator pc=injector.getInstance(PriceCalculator.class);
7      bookingPrice = pc.calculatePrice();
8    } else if(configuration == VIP_USER) {  ...  }
9  }
10 //PriceCalculator configuration
11 protected void configure() {
12   bind(PriceCalculator.class).to(LowSeasonCalculator.class);
13   ...
14 }
15 //CalculatePrice in LowSeasonCalculator
16 public double calculatePrice() {
17   //return the calculation with a low season discount
18   return pricePerNight * Math.ceil((end.getTime() - start.getTime())↩
       /(1000*3600*24D))*(1 - lowSeasonDiscount/100);
19 }
20 //CalculatePrice in RegularCalculator
21 public double calculatePrice() {
22   return pricePerNight * Math.ceil((end.getTime() - start.getTime())↩
       /(1000*3600*24D));
23 }
```

Snippet 3.1: Dependency injection implementation of the web booking application [184].

run-time binding of the behavioral adaptations. Behavioral adaptations must be decomposed according to multiple localized points in the application, similarly to hot spots in Object-Oriented Programming frameworks. Moreover, at most one behavioral adaptation can be activated per localized point. However, in the general case of Dynamically Adaptive Software Systems it would be desirable to combine multiple adaptations.

The use of dependency injection for dynamic behavior adaptation of software systems poses some difficulties. Similarly to the problems observed for the design patterns, behavior adaptations must be foreseen by programmers. In order to provide new behavior adaptations, a new component and its respective configuration need to be added to the system, which can become cumbersome and time consuming as the system grows. Additionally, a full component needs to be created for dedicated adaptations of every behavior, however small, making dependency injection ill suited for the definition of fine-grained behavior adaptations. Dependency injection is not designed to foster interaction between injected components. Rather they are supposed to stand alone for each independent request. Composition of behavioral adaptations is thus not possible using dependency injection, leading to code duplication between different components.

Service-oriented architecture

A Service-Oriented Architecture (SOA) [59] is a middleware approach for the constructions of loosely coupled service systems. Each component of behavioral

functionality is designed and provided as an independent service of the system. Inter-service communication in a Service-Oriented Architecture takes places through a standard interface, usually using XML or Web Services Description Languages (WSDL), facilitating the interchange of services for other similar services when appropriate. The flexibility to adapt the behavior of services is only restricted by the requirement that the respective WSDL complies with each of the interacting services. In SOA adaptations are defined at a coarse-grained level. The drawback of such approach is that only complete services can be adapted.

In recent years different approaches have been proposed allowing the dynamic interchange of services in SOA. Mashups are a technique for creating web service hybrids by adding value to services offered to users by associating the requested service with other complementing services. To choose the particular services to be used in a mashup, information about the user's current situation (gathered from available sensor networks) is used [24]. Using so called context-aware mashups, services are offered according to the available services and not only whenever all of the services defined for the mashup are available. Mashups could also be used to provide user customization, offering dedicated mashups to sets of users based on their surrounding environment. Cloud computing [8] has become increasingly popular as a service provider infrastructure where storage and computation services are leased to users. Web services are automatically scaled up and down according to particular situations in which the service is being used (e.g., user load, available computation resources). Cloud computing has opened the door to a new trend in computing known as Software-as-a-Service (SaaS) [183]. In the SaaS model, software products are not provided to end users as off-the-shelf products, but accessible services. SaaS is used in combination with cloud computing to provide highly scalable services to users. It also uses mashups techniques to provide added value and customized services to different users. SaaS systems envision the delivery of one single customizable service to their users. Customization is usually managed in SaaS by means of dependency injection or other software architectures that embrace behavioral adaptations [184].

Figure 3.3 shows an overview of the components of a SaaS architecture. SaaS systems are supported by a back-end in the cloud, where storage space and resources are allocated as needed. The SaaS system itself can be configured as the aggregation of other software services, for example using mashups, and client specific customizations provided by means of a customization module, for example using dependency injection or dynamic proxies. Finally the system is delivered to the final user as a web application running in a web browser or as native (web) App.[3]

As mentioned previously, software adaptation approaches like dependency injection face challenges supporting adaptation interaction —that is, applying multiple adaptations at a single application point. This problem translates to the impossibility of composing adaptation behavior in SaaS systems. Addition-

[3]Web applications that look and behave like native Mobile applications.

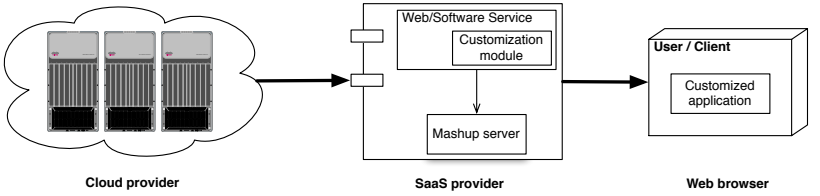


Figure 3.3: SaaS component architecture overview [8, 24].

ally, complete services must be modified for the adaptation of localized behavior, hence, we discard SOA as an ideal for the implementation of Dynamically Adaptive Software Systems.

Event systems

Event systems [188] provide a middleware platform to ease the programming of systems that reify interaction (between system components, or with the user) as the core concern of the system. Event systems allow the definition of *events* in terms of *event conditions* and *event behavior*. Event conditions dictate the state of the system (e.g., sensor information, or state variables) in which the event should take place. Event behavior defines the actions that are executed whenever the event conditions are satisfied. The behavior associated with an event is automatically executed as soon as the corresponding event conditions are satisfied. These characteristics of event systems set them as a candidate to support dynamic adaptations. However, these systems also present drawbacks for defining adaptations at different levels of granularity, and for defining adaptation events independently of the base system.

Event systems have been used to enable adaptive behavior of software systems as a response to external events or inter-component interaction [92, 121]. As an example of how an event system can be used to develop dynamic adaptive software systems, we consider the user detection system embedded in home automation application. The example is based on INI, a programming language that facilitates dynamic adaptation through event reconfiguration [121]. Snippet 3.2 shows the definition and use of events and their reconfiguration in INI. User-defined events have the structure described in Lines 4 through 9. Events are bound to an *event definition* by their name, list of input variables, and list of outputs, as shown in Line 2. Events can be accessed by describing the situations that synchronize with a particular event, giving the event definition binding (possibly preceded by its id) and the initial values of the parameters. Lines 12 through 16 show the dynamic reconfiguration of an event in response to a particular set of conditions. In this case, the conditions refer to the detection of an unknown user in the room while the house is empty. The event is then reconfigured to capture information more frequently and take pictures of the user. Such information can be used to report the presence of a burglar to the authorities. Events can be directly executed as shown in Line 19 for the

case in which there is a problem with the vital signs of the user.

```
1  //Event definition binding
2  @userDetection[frequency:Integer,imaging:Bool](Position, User, ↩
       VitalSigns) => "ini.ext.events.UserDetection"
3  //UserDetection event definition
4  public class UserDetection extends ini.event.Event {
5    Thread objectDetectionThread;
6    @Override public void eval(final IniEval eval) {
7      //generic event behavior definition
8    }
9  }
10 //event used to detect a user
11 $(burglar,emergency) detection:@userDetection[frequency=1,false](pos↩
       , user, signs) {
12  case {
13    user.unknown() && house.isEmpty() {
14      stop_event(detection)
15      reconfigure(detection, [frequency = 0.5, imaging=true])
16      restart_event(detection)
17      execute(burglar, pos,true)
18    } signs.critical() {
19      execute(emergency,user,pos)
20    }
21  }
22 }
```

Snippet 3.2: User detection and information retrieval system.

The user detection example for the home automation application confirms that event systems can be effectively used for the development of Dynamically Adaptive Software Systems. However, such development has three main drawbacks. First, in event systems the control flow of the application is driven by the input of external events rather than specified by the programmer. This is commonly known as the inversion of control problem, which can make maintenance and debugging of the system difficult [81]. Second, as a consequence of the inversion of control problem, event systems typically use *hook* methods or *callbacks* (e.g., Lines 19 or 11 in Snippet 3.2), which can turn the definition, use and maintenance of event systems into cumbersome and time consuming tasks. Third, event combinations and interaction must be manually specified within (or by interleaving) callbacks (e.g., Lines 12 through 21). As systems grow larger, more conditions need to be added and combined into callbacks, which become monolithic pieces of behavior, making the management of event systems cumbersome and time consuming.

Event systems provide a good framework for the timeliness and compatibility requirements of behavioral adaptations. However, event systems are rarely aware of their surrounding execution environment, usually they can only react to events that were defined beforehand. Upfront definition of all possible events that the system could respond to is unfeasible. Furthermore, event systems only manage one level of granularity, normally behavior responding to the events, restricting the type of adaptations allowed in the system.

Section 10.4 discusses how the formal basis for the development of Dynamically Adaptive Software Systems presented in this dissertation could be used in the setting of event systems.

Self-adaptive systems

Self-adaptive systems [117], also know in the literature as autonomic computing or self-managing systems, are conceived as closed-loop systems with feedback from *self* and the *context* —that is, from the whole software system, and the surrounding execution environment that affects the system' properties and behavior. The cornerstone of self-adaptive systems is that their life cycle should not be stopped once the system is running. The system should effectively respond to changes in its surrounding execution environment by improving its functionality or performance for the situation at hand. Self-adaptive systems, as a whole, satisfied the requirements of Dynamically Adaptive Software Systems (**D.1** through **D.5**). However, there is no single implementation of a self-adaptive system that satisfies all requirements.

In order to respond to changes in the surrounding execution environment, self-adaptive systems must provide certain characteristics known as *self-** properties.[4] Self-* properties can be categorized in three levels [159]:

- The general level is identified by two main properties, *self-adaptiveness* (consisting of self-managing, self-governance, self-maintenance, self-control, self-evaluating) and *self-organizing* (emphasizing emergent functionality and decentralization). General level properties are exhibited by systems with interactive components which have partial (or no) knowledge about the global system.

- The major level consists of the *de facto* properties of self-adaptive systems. The following properties are part of the major level: *self-configuring* (the capability of automatically and dynamically responding to changes in the environment), *self-healing* (the capability to discover and diagnose potential problems, also known in the literature as self-repairing or self-diagnosing), *self-optimizing* (the capability of managing performance and resources, also known in the literature as self-tuning or self-adjusting), and *self-protecting* (the capability of detecting and recovering from security breaches).

- The primitive level is identified by two main properties: *self-awareness* (the capability of reasoning about self), *context-awareness* (the capability of reasoning about the context).

The design of self-adaptive systems requires the consideration of four main characteristics [159]:

Adaptation unit: Adaptations can take place for different entities of the system, at different granularity levels, and to different scope extents. The entities that can be adapted as a response to changes in the environment depend on the abstraction level allowed by the underlying technology. Examples of adaptable entities include: methods, services, components,

[4] `http://www.research.ibm.com/autonomic/overview/elements.html`

or parameters. The scope of adaptations is measured in terms of their impact on the system. For example, parameter adaptation (referred to as *weak adaptation*) or the adaptation of complete system components (referred to as *strong adaptation*).

Realization strategy: Adaptations can be realized *statically* or *dynamically*. Dynamic realization of adaptations consists of a defined set of rules allowing adaptation of the system at run time. Management of adaptations can be *internal* or *external*. In the external management model, the system is composed of an adaptation engine and the software it adapts. Using an external management approach facilitates the reusability of the adaptation rules and logic. Adaptation logic is often realized by engineering the logic into the application architecture. This method is referred to as *making* adaptation. An alternative method is to use adaptive learning, in which artificial intelligence is used to render the adaptive behavior. This method is referred to as *achieving* adaptation. The realization strategy additionally comprises how generic and open the adaptation model is.

Adaptation time: Adaptations can take place *reactively* or *proactively*, respectively depending on whether or not adaptations take place due to a change in the surrounding execution environment, or whether the system can predict when a change must occur. Additionally, adaptations may take place based on a continuous stream of data (in which the system changes whenever the gathered data changes), or defining specific monitoring points in which data is gathered and analyzed to identify and respond to anomalies.

Adaptation interaction: Adaptations inevitably interact with each other and other agents (e.g., users, software systems). This characteristic concerns the *automation* level of the interaction between agents and the level of *trust* between interacting entities.

Self-adaptive systems present a high degree of flexibility and dynamicity for the realization of Dynamically Adaptive Software Systems. However, the development of such systems requires addressing some challenges for the realization of sound Dynamically Adaptive Software Systems [118, 159]. We start from the observation that currently there is no single programming model for the development of self-adaptive systems. These systems are developed, in their majority, using external frameworks usually only addressing one particular self-* property. Self-adaptive systems should provide support for all self-* properties. Additionally, most self-adaptive systems provide a static mechanism for the decision of which behavioral adaptations to take as changes are perceived in the surrounding execution environment. This raises questions about how to ensure that behavioral adaptations be consistent and provide a behavior as expected. Such a problem can relate to the problem of defining policies for the management of adaptations at run time.

Section 10.3 discusses how the formal basis for the development of Dynamically Adaptive Software Systems presented in this dissertation could be used

in the setting of self-adaptive systems.

Conclusion

Middleware solutions present two major drawbacks. First these solutions normally offer adaptability at a fine-grained or coarse-grained granularity level but not both. Second, these solutions normally interleave the definition of adaptations and adaptation logic within the base behavior of the system. This hinders the flexibility of the system as it evolves. Nonetheless, middleware solutions mostly focus on the reactivity of systems, so adaptation timeliness is ensured using such solutions. Actually, self-adaptive systems, as a whole, provide support for timeliness, granularity, flexibility, compatibility and independence. However, we note that there is no single implementation of self-adaptive systems that enables all five requirements.

3.1.3 Language Solutions

To round up the design space of Dynamically Adaptive Software Systems, the category of language solutions explores programming paradigms that provide language facilities to empower and ease the definition and introduction of adaptive behavior in software systems.

Metaprogramming

Many programming languages offer a Metaobject Protocol (MOP) to programmers, that is, a set of features to provide a way to, within the scope and limitations of the language, reason about and modify its run-time properties [110]. The features offered to programmers are *introspection* or *reflection*, to observe the state and properties of the system, and *intercession*, to catch and alter the state and behavior of the system. Using MOP capabilities provided by a language is often referred to as *metaprogramming*. Metaprogramming can be very useful for the dynamic modification of a system's behavior. However, this technique only allows gathering information about the internal state of the system. In order to reason about the surrounding execution environment it is necessary to change the base behavior of the system.

How and to what extent the MOP of a certain language could be used to attain dynamic adaptive behavior of a software system depends on the particular reflective API offered by the language. Here, we present an example of how the metaprogramming facilities of Objective-C,[5] for example, can be used to define behavioral adaptations and introduce them dynamically at run time.

One possibility to introduce adaptive behavior at run time using the MOP is inspecting the system state to check if the conditions for which an adaptation is defined are valid (introspect the state of the system at run time). For those

[5] We use Objective-C because the work of this dissertation is developed in this language. However, similar implementations can take place in other languages like Smalltalk or the Common Lisp Object System (CLOS).

cases in which an adaptation should be applied, messages should be redirected to the appropriate adaptive behavior (intercession of message calls and dynamic message re-invocations).

As an example let us take the web booking application (Section 2.3.2). The basic behavior in this example is to book hotels at a full rate. Calculation of the booking price is done via the `calculatePriceMethod:`. Let us suppose that behavioral adaptations and the base behavior are defined in an `Adaptations` class and the standard interface for the `calculatePrice:` method is empty. Such adaptations are enabled according to the state of the booking, whether it is done for the low season, or for a VIP user. The definition of these methods is shown in Snippet 3.3, in Lines 2 through 12. Note, however, that three methods are accessed using the same interface, that is, calling the same `calculatePrice:` method. To do so, we take advantage of the *method forwarding* facility available in the MOP of Objective-C. Method forwarding works by providing an implementation of the `forwardInvocation:` method in the class responsible for responding to the `calculatePrice:` base behavior.[6] The `forwardInvocation:` method, defined in Line 14, is called every time a message not understood by the class is received. Inside this method, we can reason about the state of the system to know to which method the system should respond. Lines 15 and 16 take the state variables we are interested in, and use their values. Lines 17 through 19 show how the decision of which method to apply is made.

```
1  //Adaptations class
2  - (double) calculatePriceBase: {
3    return pricePerNight;
4  }
5  //calculatePrice when for VIP user bookings
6  - (double) calculatePriceVIP: {
7   return pricePerNight * Math.ceil((end.getTime() - start.getTime())↩
        /(1000*3600*24D))*(1 - VIPDiscount/100);
8  }
9  //calculatePrice when booking low season
10 - (double) calculatePriceLowSeason: {
11   return pricePerNight * Math.ceil((end.getTime() - start.getTime())↩
         /(1000*3600*24D))*(1 - lowSeasonDiscount/100);
12 }
13 //Price managing class
14 - (void) forwardInvocation:(NSInvocation *)anInvocation {
15   IVar lowSeason = class_getClassVariable(Adaptation, "lowSeason");
16   IVar vip = class_getClassVariable(Adaptation, "vipUser");
17   if(lowSeason.value == true) {
18     objc_msgSend(Adaptation, @selector("calculatePriceLowSeason"));
19   } //Similar cases for methods calculatePriceVIP and calculatePriceBase
20 }
```

Snippet 3.3: Definition and use of adaptive behavior of the mobile file sharing application using metaprogramming.

Adapting the behavior of software systems can successfully be done via the MOP of the language. However, defining and introducing behavioral adaptations in such a way, can become quite cumbersome. As the application grows and

[6] This technique can also be used in Smalltalk by means of implementing the `MessageNotUnderstood` method.

more adaptations are defined for a particular method, the `forwardInvocation:` method becomes monolithic and harder to maintain, especially, if the different methods must be adapted with respect to a particular state of the system. The use of the MOP provides programmers the liberty of composing different adaptations by forwarding messages. However, this needs to be done manually by programmers, which has two main disadvantages: First, manually composing behavior is error-prone and difficult to maintain. Second, manual composition of adaptations hinders the dynamicity of the approach, because composition of behavior needs to be defined statically in the source code, and thus adaptations will always be composed in the same fashion. This encumbers the dynamicity and flexibility requirements of Dynamically Adaptive Software Systems.

Reactive programming

Reactive programming is a programming paradigm for the development of event-driven systems allowing the definition of events that continuously change over time [10]. In particular, reactive programming is proposed as a solution that tackles the problems of inversion of control and callback management existing in conventional event systems. Reactive programming eases the development of event systems by enabling the definition of system behavior, and automatically taking care of the execution of such behavior whenever appropriate (e.g., whenever a change is perceived). In response to such changes the behavior of the system could be adapted automatically, reducing the complexity of developing and maintaining event systems with callbacks. Reactive programming facilitates the introduction of fine-grained behavioral adaptations as a response to changes of reactive values. However, these changes need to be known statically in order to provide the respective behavioral adaptations.

Reactive programming languages are mainly characterized by two concepts: *behaviors* and *events*. Behaviors are first-class entities to represent continuous change over time. Events are first-class values describing discrete event occurrences over time. Similar to event systems, reactive programming can be used for the development of Dynamically Adaptive Software Systems. As an example we rework the user detection system in Snippet 3.4, by using Flapjax [133], a reactive programming language embedded in JavaScript.

Unlike with event systems, the complexity of managing event callbacks is reduced by the automatic execution of event dependent behavior provided by reactive languages. For example, variables at Line 1 (an event changing discretely over time) and Line 2 (a behavior changing continuously over time) are automatically updated whenever they change (or according to specific timestamps). Likewise, every function that depends on any of these variables is recalculated whenever their values change. However, the complexity of managing the conditions for the execution of specific behavior persists. This is shown in the `userDetection(..)` function at Line 4, which plays the role of a manual method dispatcher. Whenever a user is detected by the movement sensors, one of the functions defined at Lines 19 through 21 are automatically called according to the specific situation of the user. Managing such complexity can

```
1  var userE = extractEventE(m_sensor, "movement-detected");
2  var houseB = extractValueB("house-state");
3
4  function userDetection(userE, houseB) {
5    var user = db.transaction(function tx) { tx.executeSql(...); }
6    var posB = startsWith(userE, user.getPosition());
7    if(user == null && houseB) {
8      var userPhotoB = startsWith(null, "photo");
9      burglar(user, posB, userPhotoB);
10   } else {
11     var signsB = startsWith(userE, user.signs());
12     if(signsB.critical())
13       emergency(user, posB, signsB);
14     else
15       userRoomAdaptation(user, posB);
16   }
17 }
18
19 function burglar (user, pos, userPhotoE) { ... }
20 function emergency (user, pos, signsB) { ... }
21 function userRoomAdaptation(user, pos) { ... }
```

Snippet 3.4: Reactive implementation of the user detection system.

be cumbersome as a dispatching function like the one at Line 4 is needed for every adaptable behavior. Moreover, as systems grow larger, these functions undergo a combinatorial explosion of the conditions for which specific behavior must be applied, due to interaction between adaptations, but also to new adaptation conditions being introduced into the system. For these reasons we argue that reactive programming is not an ideal programming paradigm for the development of Dynamically Adaptive Software Systems.

Section 10.5 discusses how the formal basis for the development of Dynamically Adaptive Software Systems presented in this dissertation could be used in the setting of reactive programming.

Aspect-Oriented Programming

The Aspect-Oriented Programming (AOP) paradigm tackles the problem of modularizing behavior that cuts across the base modules of the system [111]. When a functionality of the system cannot be cleanly modularized within the main structural composition of the system (e.g., hierarchical decomposition in OOP languages), such functionality is said to be a *crosscutting concern* or an *aspect* of the system. Moreover, when different crosscutting concerns are present in the system interacting with each other (and with the base functionality of the system), they are said to be *tangled*. AOP provides a structured way to modularize system functionality such that behavioral adaptations do not cut across, or that is tangled with that base functionality. AOP provides a modular approach for the definition and introduction of behavioral adaptations. However, this approach poses limitations with respect to the composition between aspects and their respective behavioral adaptations, which can give rise to behavioral inconsistencies.

One of the most widely used mechanisms used in AOP to address the prob-

lems of tangled functionality and crosscutting concerns is the use of *advice* and *pointcuts*. Advices specify whether the aspect's behavior precedes, succeeds, or replaces the behavior specified in the base system. Using advices, different localized behavioral concerns in the system can be either extended (by means of preceding or succeeding advices), or completely concealed (by means of the replacement advices). Pointcuts are a declarative way to select different *join points* in the system's execution. Each advice is defined alongside a pointcut specifying the points of the program execution in which the advice is joined with the base behavior of the system. The mechanism of pointcuts and advices can also serve as a mechanism to introduce behavioral adaptations into the system, for example, if we use *replace advices*. However, application of advices into the base functionality of the system according to pointcuts is not always done at run time, but at *aspect weaving* time. Different mechanisms for dynamic aspect weaving have been proposed. CaesarJ [7] is an AOP language that allows dynamically scoped activation of program definitions. Nonetheless, declaration of pointcuts is performed statically, making it ill-suited for the maintenance of multiple advices. JAsCO [181] is an aspect language tailored for the development of component-based software. JAsCO enables the runtime application and removal of aspects by means of *connectors*. Connectors specify the context and order in which aspects are deployed into the system. Every time the context conditions specified in the connector are satisfied, its defined aspects are deployed into the system and are applied in the order of their definition. Ordering of applicable aspects addresses the feature interaction problem [28]. PROSE [149] is an AOP system introducing dynamic aspect weaving. Aspects can be woven and unwoven at run time through an *aspect extension manager*. Dynamic aspect weaving can be used, for example, for adaptation of services in response to event changes in the environment. Dynamic aspect weaving fosters system flexibility, for instance, the ability to express join points that capture only certain invocations of a given method, or application of aspects to particular contexts of execution.

Take for example the behavior adaptations introduced in the web booking application (Section 2.3.2). These adaptations to the booking behavior can be dynamically introduced using AOP as shown in Snippet 3.5. Line 4 shows the definition of an advice which replaces the original method implementation (this is dictated by the `METHOD_ARGS` definition). Initially the advice matches all method invocations that return a `double`. This is specialized by the pointcut defined in Line 8, which specifies to only match methods (members of any class) that are named `calculatePrice`.

Aspect instances are dynamically introduced and withdrawn via the PROSE aspect manager as it is shown in Lines 14 and 15 of Snippet 3.5. To manage interaction between aspects, PROSE offers two additional constructs. The `proceed()` construct allows to escape from the advice's execution and return to the base functionality. `proceed()` is often used to extend the behavior of the base system by performing additional computation, and then continue with the regular flow of the system (i.e., without aspects woven into it). It might

```
1  class LowSeasonDiscountAspect  extends Aspect {
2    private static double lowSeasonDiscount = 20;
3    Crosscut lowSeason = new DiscountCrosscut() {
4      public double METHOD_ARGS() {
5        return pricePerNight*Math.ceil((end.getTime()-start.getTime())↩
             /(1000*3600*24D))*(1-lowSeasonDiscount/100);
6      }
7      //specialization matches the calculatePrice method
8      protected abstract PointCutter pointCutter() {
9        return (Within.method("calculatePrice"));
10     }
11   }
12   //Aspect insertion and withdraw
13   LowSeasonDiscountAspect asp = new LowSeasonDiscountAspect();
14   Prose.getAspectManager().insert(asp);
15   Prose.getAspectManager().withdraw(asp);
16 }
```

Snippet 3.5: Implementation of the web booking application using PROSE.

be the case that multiple aspects can be applied to a particular join point. The `setPriority(int)` construct is introduced to resolve the order in which aspects are applied. Aspects defined with a lower priority are applied before those with a higher priority. If no priority is defined, it is assumed that aspects have a priority of 0. Whenever multiple aspects are applicable to a particular join point, the use of `proceed()` does not immediately escape from the aspect execution to the base level, but rather, the aspect escapes to the next applicable aspect according to their ordering.

AOP, and in particular the dynamic weaving incarnation of AOP, provides the necessary functionality to allow dynamic adaptation of system behavior in response to changes in the system's environment. However, two major challenges can be identified. Aspects are not meant to be composable units of behavior. Behavior provided by aspects is devised to adapt (and maybe reuse) the behavior defined in the base application. When multiple aspects (with the same priority) are applied to a same join point, the order in which they execute is usually non-deterministic —that is, the order in which aspects execute can change from one execution to the next.[7] The second problem AOP poses for applying dynamic adaptation of software systems is that aspects are woven into every join point matching its pointcuts. There are particular situations in which we do not want to weave the aspect for a particular point-cut, while in all other situations the aspect should be woven. This problem is eased by the introduction of an aspect manager in dynamic aspect weaving. In such a case we recognize that the management of the conditions for inserting or withdrawing an aspect can become complex as the number of aspects and special situations increases, possibly leading to inconsistencies in the observed behavior. Therefore, AOP is not an appropriate paradigm for the development of DASS.

[7] This is the case of AspectJ, other AOP languages, such as JAsCO, use a defined ordering for the application of aspects, based on the order in which they are declared. However, this ordering may not be desired for all possible cases of interaction between aspects.

Context-Oriented Programming

Context-Oriented Programming (COP) [90] is an emerging programming paradigm similar to the AOP paradigm, addressing the modularity problem of separating the base behavior of a system from its adaptive behavior. This paradigm provides a high level of dynamicity for the introduction and withdrawal of fine-grained behavioral adaptations. The COP paradigm also addresses the problems of compositionality and interaction of behavioral adaptations. For these reasons we chose COP as our experimentation platform in the setting of Dynamically Adaptive Software Systems. An extended introduction and explanation of the COP paradigm is presented in Chapter 4.

Conclusion

In this dissertation we are interested in providing support to develop software systems that provide a fine-grained adaptability (not excluding more coarse-grained types adaptability) while enabling a high level dynamicity. As evidenced by this overview, the solutions realizing Dynamically Adaptive Software Systems that most closely approximate these requirements are those offered directly within the programming languages. In particular, one solution that seems to provide the most flexibility for the definition, introduction, and dynamicity of adaptations, is Context-Oriented Programming.

3.2 Models for Conflict Resolution and Inconsistency Management

The ability to dynamically adapt their behavior empowers software systems to become ever smarter and flexible. However, with such great power comes the great responsibility of ensuring that the system behaves "correctly" in every possible situation —that is, the observed behavior of the system at run time can be predicted during its design and development.

Section 3.1 already mentioned different situations in which unpredicted behavior could be observed, for example, whenever certain adaptations are combined or interact with each other. Adaptation interaction may give rise to inconsistencies in the system behavior (i.e., the behavior observed by the system is not as predicted). For example, in the web booking application whenever a booking is done by a VIP user during the low season, it is unclear which discount is applied to the flat rate of the booking. To avoid inconsistencies, systems must provide a way to coordinate or manage interaction and composition of adaptations at run time. In this section we explore state of the art in coordination models for software systems at run time.

Before exploring existing approaches for the coordination or management of software systems later in this section, we provide a definition for run-time models (also called execution models), alongside key criteria to evaluate their appropriateness with respect to Dynamically Adaptive Software Systems. A

model is understood as a set of formal elements describing a software system, designed to analyze particular characteristics or subparts of the system. Example analyses are: checking for completeness of the system, transformation of the model into code, or formal abstraction of the system [132]. A run-time model extends the initial definition of model to also describe dynamic aspects of the system under study. Definition 3.1 presents our definition of a **run-time model**, which is based on the definition given in Bobbio [20].

Definition 3.1. *A run-time model is an abstraction of the state of a system at run time —a snapshot of the system. A run-time model can be represented as an abstraction of the system's state space and the different paths of executing actions between these states.*

The purpose of abstracting the system's dynamic semantics by a run-time model is to allow reasoning about the dynamic properties of the system by means of analysis techniques. For example, satisfiability of invariant properties on all states, or possible reachable states at run time, can be verified by analyzing safety or liveness, or reachability of the system, respectively. However, state spaces can be unbound. The choice of the abstraction level of the model is crucial to avoid an explosion of the number of states during the system analysis.

In what follows we explore state-of-the-art approaches proposed for the management of software systems. We divide the design space of conflict resolution models for software systems into four categories according to the formalism behind each approach. These are, architectural modeling approaches, logic approaches, rule-based approaches, and state machine approaches. Each of the presented models is evaluated with respect to the interaction, safety, abstraction, and decision requirements for inconsistency management of Section 2.4.

3.2.1 Architectural Modeling Approaches

Modeling artifacts has been widely used in the software engineering process to describe systems by abstracting specifics of their implementation and providing a high-level view of the system. Normally, software models are used solely for the design of the system or for its documentation. However, more and more interest is seen in the use of models actively support the development and execution of the system. In the category of architectural models we discuss some of the existing artifacts used to model adaptive software.

Modeling transformations

Software system models as understood by the MDE community are a series of artifacts used to describe (parts of) a system. Within the modeling community a large body of research has been applied to the management of consistency among the models describing a system, and throughout the evolution of the system [178, 146]. Model artifacts are normally treated statically. Whenever the model changes, the changes are independent of the current state of the system and its dynamics. To apply changes made to a model, the system must

be stopped and re-compiled. Various approaches, however, allow the evolution of models at run time, usually by means of techniques such as aspect weaving or dynamic SPL [135].

Here we discuss a transformation-based technique for the description of Dynamically Adaptive Software Systems using models. The idea behind this approach is that the system is described by a modeling language (like UML), whereas particular parts of the system that should adapt to specific situations are described as independent models. The adaptation of the system at run time is realized by means of graph transformation rules [165, 46]. The Context-Aware Application (CAA) model proposed by Degrandsart et al. [46] defines adaptations as a multi-dimensional space representing all of the situations in the surrounding execution environment that are relevant for the system. An adaptation is an instantiation of the space by giving specific values to each dimension in an adaptation vector. Each value of an adaptation is represented by a specific model. Adaptation of the system is performed by means of a transformation function, which given an initial model and a model adaptation, produces an adapted model.

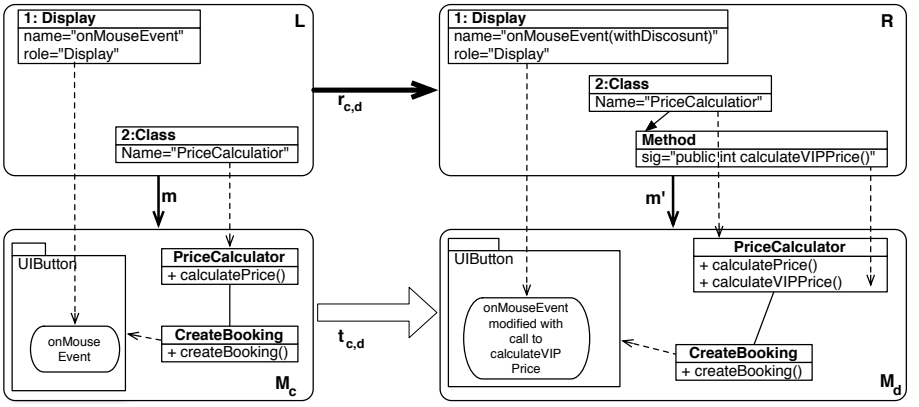


Figure 3.4: A model transformation for the web booking application.

We revisit the web booking application example using the model transformation approach introduced by Degrandsart et al. [46]. For this example our adaptation vector consists of one dimension with two possible values — the discount adaptation with values "low season" and "VIP client". Figure 3.4 shows in the lower left-hand corner the basic model of the system, identified as M_c, which provides the behavior to create a booking and calculate its price; where the behavior is triggered by a button being clicked in the user interface. The model identified by L provides the initial adaptation where there is no discount. The right-hand side of Figure 3.4 shows the adapted model after a transformation takes place. The R model adapts the event sent by the UI button and extends the `PriceCalculator` class with the addition of method `calculateVIPPrice()`. The model corresponding to this adaptation

is the model M_d. The arrows labeled m and m' represent the *match* functions linking the elements of the adaptation models to the corresponding elements of the system's model. Arrow $r_{c,d}$ denotes the transformation rule, and arrow $t_{c,d}$ denotes the actual graph transformation.

Graph transformation rules can conflict with each other, for example, when one rule deletes a model element required by the second rule as an input, or when a transformation rule changes attribute values of the adaptation vector, such that a second transformation rule is no longer applicable. The CAA model provides an analysis method to identify these type of conflicts between graph transformation rules by means of the executable language AGG [182]. The CAA model can be used to identify conflicts between transformation rules by analyzing the coverability property of adaptations defined in the system —that is, analyzing if all adaptations defined in the system can be reached from the initial state of the system by means of graph transformation rules.

The transformation-based CAA model provides support for both abstraction and decision properties, thus it can serve as a run-time model for Dynamically Adaptive Software Systems. Although the CAA model has been successfully used for the development of Dynamically Adaptive Software Systems it still has some rough edges which discourage its use. First, the system is not represented by a single model, but by a set of models (both the basic representation of the system and its adaptations). Such proliferation of models can become complex to manage as the application grows and more entities and adaptation situations are added to the system. Moreover, due to the exponential growth of transformation rules as the number of adaptation situations and their values increase, the definition of transformation rules can become cumbersome. Second, there is no way to express interaction between adaptation dimensions (or their values), this can cause an unintentional covering of one dimension by another —that is, the value space of one dimension is contained in the value space of another dimension (during the reduction and transitive closure calculation).

Feature-Oriented Domain Analysis

Features initially emerged with the goal of expressing distinct functionality of a software system. This concept of a feature is referred to as conceptual, because it only regards observable behavior of the system but not its implementation. The Feature-Oriented Programming (FOP) paradigm [151] arose to consider the implementation aspects of conceptual features. A system consists of a set of artifacts describing the program's functionality. Features encompass parts of these artifacts, distinguishing between coarse-grained and fine-grained features [108].

In the setting of FOP, software systems are usually considered as aggregation of features, describing the main modules of functional behavior of the system. To represent such modules and their sub-parts, the Feature-Oriented Domain Analysis (FODA) methodology was proposed as a modeling technique [106]. Feature models, as shown in Figure 3.5 for the case of the web booking application, can also be used to identify the different possibilities, or variations,

providing the behavior of a functional model. Additionally, feature diagrams make it possible to model the interactions between different features.

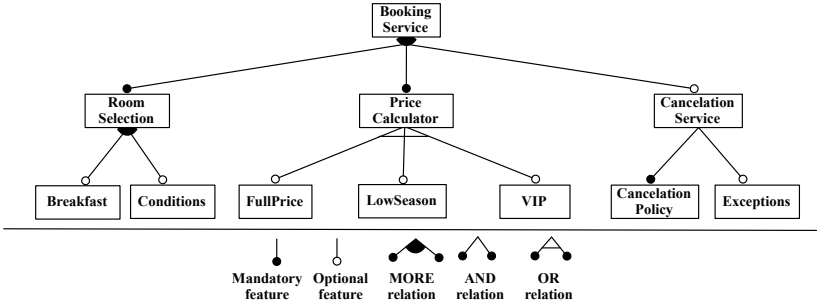


Figure 3.5: Feature model aggregation for the web booking application.

Feature models have been traditionally used in a static setting, where given a feature model, it is possible to reason about its composition and behavior beforehand. Reasoning techniques in FOP allow the identification of configurations of features that cannot occur in the composition of a software system —that is, the identification of features that cannot be deployed together in a software system.

In the setting of feature modeling the concept of feature interaction [28] was introduced with the objective of identifying inconsistencies that may arise from the activation and interaction of different features, based on the defined dependencies between them. In the general case, feature interaction is a challenging problem. This problem becomes even harder when addressed at run time. Techniques for the identification of feature interaction at run time have been developed over the years. However, for the majority of cases the problem of run-time feature interaction is undecidable. To overcome this problem, recent approaches address the dynamic feature interaction problem by the upfront definition of policy rules between features that may interact at run time [126]. Whenever a feature is included in the program at run time, the policies are verified. In case there is an interaction problem, the associated corrective process defined by the policy is applied.

Feature diagrams effectively abstract software systems and provide support to analyze system properties. However a main drawback is that policy and resolution rules need to be predefined. Therefore, they may not cover all possible interactions of features that can appear during the system execution. Moreover, verification of policies and application of conflict resolution techniques have been reported as costly operations, making these analysis techniques less suitable for highly dynamic settings.

Conclusion

Architectural approaches provide a high level abstraction for modeling software systems, presenting their main components and their principal interactions. In

the case of Dynamically Adaptive Software Systems, such abstractions of the system could be used at run time to manage the interaction between the different components of the system. Run-time interaction is normally managed by means of transformation rules or contracts which explicitly specify the allowed and disallowed interactions between the system adaptations. These approaches only allow to reason about the correctness or safety properties of interactions beforehand. To provide such support these models rely on external models or formalisms (usually using external tools as AGG or SAT solvers as Promela) that allow reasoning over different properties about the model and the interaction policies. Such additional specification burdens the development of the system, making the approaches inappropriate as models for the inconsistency management of Dynamically Adaptive Software Systems.

3.2.2 Formal Approaches

Analysis of software systems is normally supported by means of formal approaches. Software systems can be modeled by raising their level of abstraction to an appropriate mathematical formalism. Formal models are normally used at early stages of the development process to benefit from the mathematical model, using it to prove or identify interesting properties of the system.

Logic programming languages

Over the years different programming languages that implement logic formalisms, such as Prolog [58] or Soul [195], have been developed to create bridges between software systems and logic reasoning engines. The use of such logic languages allows programmers to reason about software systems by querying the system for particular properties using logic expressions.

The general way in which logic programming languages reason about system properties consists of the following steps:

1. A representation of the system to be analyzed is (automatically) extracted, for example, a model of the program's object structure, the control flow graph of the system, or the Abstract Syntax Tree (AST) of the system.

2. The property of the system that we want to analyze is expressed as a logic expression (in the logic language), over the extracted system representation.

3. The logic expression is evaluated as a query over the system representation until an answer, or a counterexample is obtained.

The added value of using dedicated query (logic) languages to analyze system properties is that it facilitates reasoning about properties that should be satisfied over a period of time (using CTL or LTL logic), or to reason about properties holding up in an entire path of execution (using regular path expressions). Logic programming languages are commonly used for analyzing system

properties. Such analysis is performed on an abstract model of the system, where the state and transitions of the system are represented, according to the needs of the properties to be analyzed.

Logic programming languages provide an abstract representation of a system, and the means to reason about system properties by querying such representation. In order to use these programming languages for the modeling and management of Dynamically Adaptive Software Systems, they must be extended to support reactivity and express interactions between program entities (similarly to the extensions required to reason over source-code). Since the languages must be extended and modified, reasoning about them must be revisited, to take into account the introduced concept. Thus, logic programming languages cannot be used straightforwardly as conflict resolution models.

Algebras and logic

Different mathematical formalisms have also been used to model and foster formal reasoning about software systems and their properties. The idea in these approaches is to make abstractions of the system within the boundaries of a specific mathematical formalism and to decide about certain system properties within such formalism. We discuss here more suitable formalisms to reason about Dynamically Adaptive Software Systems.

Modal logics [19] have been used to represent necessity and possibility conditions for system properties. Modal logics are mostly used to express temporal conditions, but they can also be used to express conditions like termination of programs, in the case of the propositional dynamic logic. Using modal logics it is easy to reason about past or future states of a system by means of modal formulas, for example, expressing concerns like "does property x holds for every state following the current state" or "has property x held at any point in the past". In the context of Dynamically Adaptive Software Systems such reasoning could be used to identify if a particular (set of) functionality is provided for all applicable adaptations of the system. For example to express situations like "will method `foo` be applicable in the future".

Process algebras [86, 52] are used to model concurrent processes, providing high-level abstractions for operations between processes such as parallel composition, communication, replication, restriction, and synchronization. The most prominent representative of process algebras is the π-calculus specification [134]. In particular, in π-calculus it is possible to express the conditions under which two processes communicate (via restrictions) which can be used to restrict how and when the main process of the system communicates with a particular adaptation. Specifically, it would be possible to compose different adaptations to a particular process. In the web booking application the parallel composition of processes is expressed in π-calculus as:

$$\{\} \vdash \textsc{Booking} \xrightarrow{p} \{p\} \vdash F|(\nu l)(\bar{p} - LSP)|(\nu v)(\bar{p} - VIPP)$$

This expression represents the request for a booking price carried through

the p channel; the price is calculated by the full price process F and can be composed with either the reduction price for low season -LSP- or a VIP user price -$VIPP$- processes whenever their restrictions apply.

Coalgebras, and in particular coalgebraic specification [97], has been used to express the dynamic behavior of systems. Coalgebraic specifications are structured using inheritance and aggregation from Object-Oriented Programming (OOP), which are used to model state-based systems (where the state is considered as a black box). Dynamic behavior can then be expressed specifying conditions in which particular behavior should take place or not, hence, specifying system adaptations. The use of coalgebraic specifications allows us to reason about dynamic behavior in terms of invariance and bisimilarity. Additionally, modal operators may be used within the coalgebraic specifications as invariants for reasoning about future states, and safety of progress/modal formulas among other system properties.

These three formalisms are used on their own as an abstract model to prove consistency or decidability properties about the systems they model. However, these models are used to represent program properties, rather than representing the actual program, (consequently they are not used at run time). To represent the program state and transitions, concrete models based on the formalisms are defined. Examples of these concrete models are: abstract state machines [23], algebraic petri nets [55], alternating automata [179] or computational tree logic [40]. Regardless of the concrete model used, verification and analysis of system properties are often done by means of model checking techniques, requiring two specifications of the system.

Model checking

Model checking is an analysis and verification technique that is orthogonal to all formal approaches described in this section. Model checking verification provides information about system properties based on a logic formulae specification of it. Model checking techniques for the verification of logic formulae include abstract interpretation, partial order reduction, or automated theorem proving. The most prominent model checking technique is based on the boolean SATisfiability problem (SAT), which is commonly used to decide if a system satisfies a set of logic formulae [150].

Model checking, and in particular SAT solvers, could be used in the setting of Dynamically Adaptive Software Systems to verify if adaptations of the system only take place as initially specified. This can be done through logic formulae describing the conditions in which adaptations and their interaction could take place. Formulae are used to express the disallowed states of the system. Each formula is verified by the SAT solver. If the formula is satisfiable, then the behavior of the system is incorrect.

In order to use model checking techniques, it is usual to describe the state space of the system by means of a state machine like approach (e.g., an automaton). System properties are verified by means of logic formulae on such state space. Using two specifications of the system (i.e., the state space and its

property formulae) could be error prone, as both representations must change simultaneously over time. Additional support should therefore be provided to automate the generation of the logic formulae from the initial state space of the system.

3.2.3 Conclusion

Formal approaches are introduced for the abstraction or verification of software systems. These usually consists of two parts, a formal specification and a verification model (e.g., a model checker). These approaches usually combine an abstract model of the system and a verification model to reason about the system, satisfying the abstraction and decision requirements. The representation of the system state by means of, for example, logic variables, also requires a way of composing the adaptations such that all allowed combinations of adaptations are taken into account. Not having such a composition mechanism would hinder the safety property of introducing new adaptations (Requirement M.2). Formal approaches provide a means to model interactions between adaptations in Dynamically Adaptive Software Systems. However, such definitions remain abstractions of the system. Additional development efforts are required to use such interactions (and other specifications of the system) at run time.

3.2.4 Rule-Based Approaches

Rule-based approaches consist of (external) production rule systems managing and coordinating the behavior of a software system. In this section we only consider rule engines based on forward-chaining. Rule engines that infer the validity of rules describing the system and take the actions described by those rules, based on the available data. We do not take into account backward-chaining rule engines, because they infer validity of rules based the goal system which in the setting of Dynamically Adaptive Software Systems may not always be known beforehand. Defining all possible configurations of adaptation (goals) to be verified can become cumbersome, making backward chaining rule engines not appropriate for modeling Dynamically Adaptive Software Systems.

Rule based engines are used as coordination models providing a consistent view of a software system. The view of the system can be generated, for example, by means of a fact space [136]. That is, by a model gathering state data of the system, or *facts*. Rules describe possible actions that can be taken by the system whenever a set of conditions becomes valid. A rule, or fact is then described as a tuple of conditions on the state of the system, and the actions to be taken.

As an example Snippet 3.6 shows the description of different facts (Lines 2 and 3) as well as two rules (at Line 5 and Line 8) describing the pricing conditions in the web booking application. The `:vipPricing` rule is used to regulate the price charged for VIP users. This rule is applicable if and only if there is a user fact in the current (fact space) session, for which its status is vip. Note that fact spaces allow us to restrict the domain in which production

rules are applicable. For example, the `:vipPricing` production rule is only applicable when the user is logged into a session (modeled as a fact space). However, it is also possible to define rules that are expressed for the totality of the system. This is the case for low season bookings. The `:lowSeasonPricing` fact would also be applicable because the `season` fact is contained in the `public` fact space.

```
1  //facts
2  season(low).
3  user(vip).
4  //rules
5  :lowSeasonPrice(?price) :-
6    public -> season(?season),
7    ?season == low
8  :vipPrice(?price) :-
9    session -> user(?statuts),
10   ?status == vip
```

Snippet 3.6: Facts and production rules for the web booking application.

Forward chaining rule-based engines are commonly implemented using the rete algorithm [65]. The rete algorithm provides a pattern matching implementation for rule-based engines. The algorithm consists of a network of nodes, where each node represents a pattern of conditions describing the different production rules. Moreover, each node also keeps track of the facts that it satisfies. Facts are propagated in the network according to their matching node. Whenever a fact satisfies all the conditions of a rule, a leaf node (a node with no successors) is reached and the action described by the production rule is executed.

Such rule-based engines provide support for the realization of Dynamically Adaptive Software Systems by introducing adaptations as actions of production rules, and by adding the facts matching the rule to the fact space, whenever the action is supposed to take place and withdrawing it when the action is not supposed to be present in the system. The type of rule-based engines presented here effectively fulfill the abstraction and safety requirements for runtime models. Even if rule-based engines ensure consistency of the system by ensuring that production rules actions only take place whenever their conditions are satisfied, it is still up to the programmer to ensure that the provided facts are correct and there is no accidental interactions or starvation of production rules. Rule-based engines do not provide support for the decision requirement (**M.4**) of conflict resolution models described in Section 2.4.

3.2.5 State Machines

State machines are commonly used approaches for the abstraction of a system by expressing its different states and actions. Normally, state machines can be represented as a graph, where nodes represent the states of the system and labeled edges represent actions that take place between different states. Here we overview some of the existing and widely use state machines approaches.

Automata & Labeled Transition Systems

Automata [94] are graph-based models used to describe system behavior based on their possible states, and the set of actions to be taken for each state. Automata are normally used to verify system properties, such as program termination. Moreover, automata are preferred as modeling techniques for software systems because the model itself is easy to operate, for example, for the merging, intersection, and parallel composition of software systems. Labeled Transition Systems (LTS) are rule-based systems describing the states and actions of a system. Since LTS are normally represented by means of automata we consider them jointly in the following.

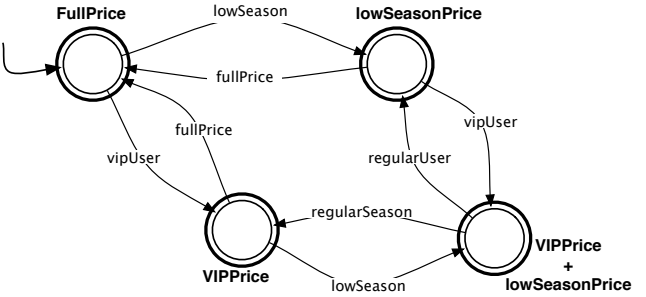


Figure 3.6: Automata model for the web booking application.

Figure 3.6 shows an automata representation of the adaptations in the web booking system. In Figure 3.6 states are represented as circles, actions are represented as labeled arcs between states, final states are represented as double-circle states, and the initial state of the system is represented as the arrow adjacent to only one state.

Representing dynamic behavioral adaptations of a system by means of automata focuses on one particular state of the system at a time —that is, only one of the states of the automaton is "*active*" at a time. This means that it is required to express all possible combinations of states in the system, leading to an explosion in the number of states as the system grows. As it is possible to see in Figure 3.6, it is possible to reach all states from any other state defined in the web booking application, making the automata cluttered and difficult to manage. To manage such complexity automata are often expressed as the product of simpler automata.

Statecharts

Statecharts formalisms provide a specification of the internal behavior of system components as well as the interactions between components in the setting of reactive and event-based systems [83]. Statecharts are an extension of state transition diagrams (e.g., automaton and LTS) proposed to tackle the state explosion problem. Statecharts model systems by successfully modularizing their

various components in a hierarchical way. Additionally, the model enables us to easily express general properties such as *clustering*, *orthogonality*, or *refinement*.

Figure 3.7 shows the example of a statecharts diagram describing the user detection system in the home automation environment, where states are represented as rounded rectangles, events are represented as labeled arcs between states, and conditionals are represented as circle elements. Figure 3.7 illustrates these properties of statecharts for the home automation application (Section 2.3.1). Clustering is depicted by the containment of the different states within each other, the hierarchy of the states is represented by the containment relation, inner states have a higher hierarchy than outer states. Orthogonality is depicted between the `Detect user` and `Emergency` states by means of the dotted line separating the two states. Whenever the container state is accessed, the two sub-states work independently of each other, by applying the `User adaptations` state in the first case, and the `Monitoring` state in the second case. Refinement is the property of statecharts to capture the different components of which the system consists of. For example, Figure 3.7 represents one of the components (the refinement) of the complete home automation system.

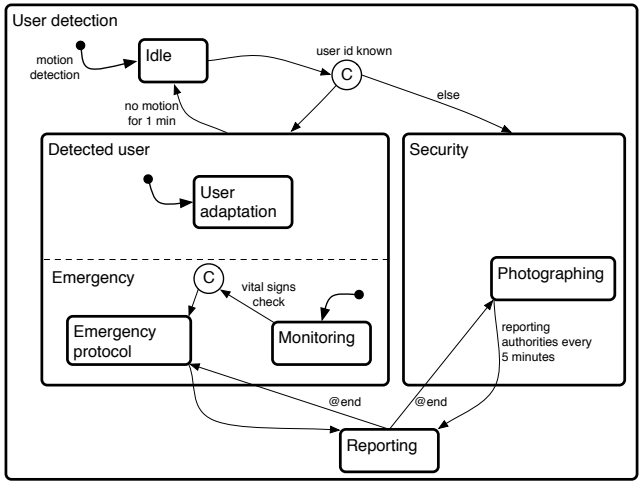


Figure 3.7: Statecharts model for the user detection service.

In addition to the visual representation of the system, statecharts models also offer a formalization and analysis of the system [120]. Together with the formalization, statecharts enable the abstraction of software systems by giving a semantic meaning to the model layout, for example, to express clustering or orthogonality properties. These properties cannot be expressed with other formalisms as automata. Moreover, since the model can be easily extended, more complex systems can be expressed without raising the complexity of the model or yielding a state explosion problem. For example, it is possible to express timeouts or temporal notions in the execution of events, as shown in

Figure 3.7 with conditions as "no motion for 1 min". Regarding the decision power, via their formalization, statecharts offer the possibility to analyze the system beforehand by means of model checking techniques, since statecharts explicitly model the system behavior and state changes as reactions to event changes. In order to satisfy the safety requirement, statecharts have to be extended to by, for example, adding support to introduce new states via event triggering.

Dataflow graphs

Dataflow programming [100] originated from the exploration of parallel systems. The motivation behind dataflow programming is that programs are represented as directed graphs where data flows between nodes along the graph's arcs. Nodes of the graph represent program entities (e.g., modules, objects, instructions according to the level of abstraction used). Directed arcs of the graph represent data dependencies between program entities. Dataflow graphs are used to specify the computation of the system (i.e., *are* the program), as well as its coordination or management.

Figure 3.8 shows how the movement detection service example could be expressed as a dataflow graph, for the example we take inspiration in the way dataflow graphs are defined in AmbientTalk/R^V [129]. The management model of dataflow systems consists of *operators* processing input data (e.g., node `RFIDSensor` processing the `movementDetected` event in Figure 3.8) and producing output data (e.g., node `HouseEmpty` producing the `empty` boolean value in Figure 3.8). Dependencies between nodes are explicitly represented by edges in the graph. Data always flows from the outputs generated by a node to the input of another node. A node in a dataflow graph is said to be enabled to fire if there are values available for all of its inputs. Node fire (execution) can take place at one of two moments. Once *all* of its inputs have received a (new) value, or whenever *one* of its input values changes. These two approaches are referred to as *synchronous dataflow* and *asynchronous dataflow*, respectively.

Arcs for nodes like `RFIDSensor` are called forking arcs. Whenever data reaches a forking arc it is duplicated and sent to each of its subsequent nodes. Dataflow graphs enable the parallel execution of processes. As an example, Figure 3.8 depicts the tasks that can be computed in parallel by the H_i doted lines. All operations in the same line can fire in parallel (as long as they can fire). That is, nodes `UserIdentification` and `MonitorUser` can be computed in parallel if `RFIDSensor` produces output data.

Dataflow graphs have been successfully used as visual programming languages, where the dataflow graph is not only a representation of the system, but it is indeed the execution model of the system. Moreover, dataflow graphs are also successfully used as management modules of the system, meeting the modeling power criteria for run-time models, as it is the case in AmbientTalk/R^V. Unfortunately, dataflow graphs do not provide a direct way to analyze properties of the system. However, some dataflow programming languages, like NL, allow using the dataflow graph itself as a visual debugger [84].

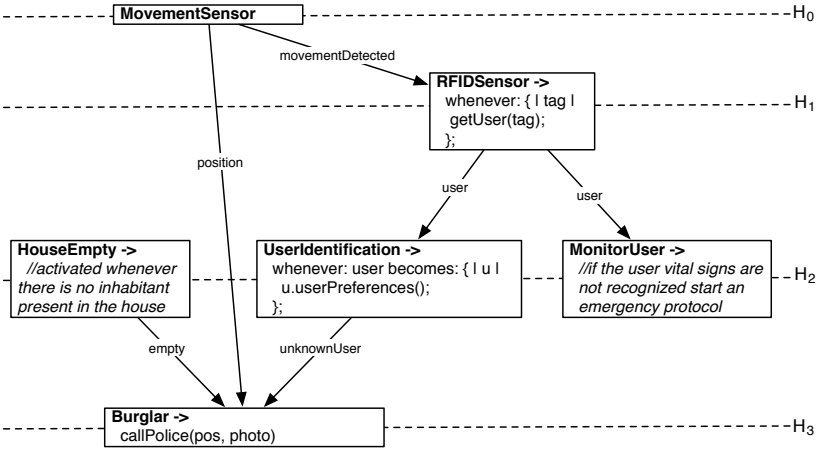


Figure 3.8: A dataflow graph for the user detection system

Petri nets

In contrast with the different approaches mentioned before, an approach that fulfills all the requirements for conflict resolution models described in Section 2.4, among other interesting properties, is the Petri net model [137]. Petri nets are used to model the states and dynamics of a system. Petri nets combine the modeling power of automata and dataflow graphs allowing us to define adaptations as particular states (places) of the system, satisfying the abstraction requirements.[8] Interactions between them through the firing of actions (transitions) follow the operational semantics of Petri nets, satisfying the interaction requirement. New adaptations could be introduced by means of Petri net composition, satisfying the safety requirement. Additionally, Petri nets provide a series of analyses that allow us to reason about different system properties, satisfying the decision requirement. Since Petri nets satisfy the four requirements for inconsistency management models, we decided to use Petri nets as our platform for the modeling of Dynamically Adaptive Software Systems. An in-depth description about Petri nets is given in Chapter 5.

Conclusion

This section described different models and formalisms that could be used as run-time support for Dynamically Adaptive Software Systems. We argue that state machine approaches, and in particular Petri nets, are among the most appropriate for the representation and run-time execution of Dynamically Adaptive Software Systems, since they satisfy the requirements **M.1** through **M.4**

[8] Note that in Petri nets there is no state explosion problem,a s its inherent support for concurrency allows to model different states being active simultaneously by means of the Petri net marking.

described in Chapter 2. In addition to the abstraction and decision properties that the majority of these approaches have, we also took other criteria into consideration to serve as inspiration for the modeling and specification of Dynamically Adaptive Software Systems. These criteria are:

- Having a single representation for modeling different aspects of the system.
- Having a well-defined behavioral semantics of the system, generality for the modeling of different types of systems.
- Being able to visualize the system, in particular, the interaction between adaptations and possible actions over adaptations.
- Being able to support for interactive simulations.

Although it would also be possible to extend the other approaches presented in this section in order to satisfy all requirements for inconsistency management models, taking into account all these characteristics, we believe out Petri nets to be the most appropriate approach for managing inconsistencies in Dynamically Adaptive Software Systems.

3.3 Conclusion

This chapter presents an overview of Dynamically Adaptive Software Systems from two different viewpoints, serving as background to our end goal of developing software systems that are highly dynamic and predictable. The first viewpoint, presents the state-of-the-art in implementation techniques of dynamic adaptations to a system's behavior. The second view point, presents various models used for conflict resolution in Dynamically Adaptive Software Systems.

Table 3.1 provides an overview of the surveyed approaches with respect to the requirements defined in Section 2.4. For each of the surveyed approaches we take into account all the requirements, this is due to the fact that some approaches indirectly provide support for the requirements specified for both categories —that is, for Dynamically Adaptive Software Systems and conflict resolution models. Supported requirements for each approach are marked as black cells in Table 3.1. Requirements which are satisfied partially, or that are commonly satisfied by the combination of the studied model with another model are marked as gray cells. unsatisfied requirements are white cells.

D.1 Timeliness: The timeliness requirement refers to the ability of software systems to adapt their behavior, as a reaction to changes in the information gathered from their surrounding execution environment —that is, how promptly is the adapted behavior observed in the system with respect to its surrounding execution environment.

All surveyed solutions focus on the prompt composition of behavioral adaptations with the base behavior of the system. The only solution not fully providing a timely application of adaptations is dynamic software upgrades. Dynamic software upgrades may not be prompt because, the system must reach a *quiescence* before applying the system upgrades.

Both, architectural and language solutions are appropriate for the development of timely Dynamically Adaptive Software Systems.

D.2 Granularity: The granularity requirement refers to the ability of software systems to adapt the behavior of various program entities such as variables, objects, processes, components, and the like.

Adaptations are mostly handled at a fine-grained or coarse-grained level, but not both. In general, architectural and middleware solutions offer the possibility to adapt the system entities at the level of granularity the solutions were envisioned for. Architectural solutions provide adaptation of coarse-grained entities such as objects or components, and middleware solutions provide adaptation of fine-grained entities such as methods. Nonetheless, we observe that language solutions support the definition of adaptations at both levels of granularity. In language solutions is possible to adapt methods, objects, or components according to the requirements of the surrounding execution environment.

Language solutions are then more appropriate for the development of Dynamically Adaptive Software Systems with adaptations taking place at different levels of granularity.

D.3 Independence: The independence requirement refers to the characterization of adaptations without interfering with the base functionality or other adaptations defined in the system.

For most of the solutions, adaptations can be defined independently of the base behavior of the system. In middleware solutions adaptations can be defined independently of the base system, however, these solutions require modification of the base system to specify the places in the system in which adaptations can be composed with the system. Language solutions, on the other hand, preserve independence between adaptations and the base system. Language solutions use dedicated abstractions to specify how and where are adaptations composed with the system without having to modify its base behavior.

Language solutions are then more appropriate for the development of independent adaptations in Dynamically Adaptive Software Systems.

D.4 Compatibility: The compatibility requirement refers to how faithful software systems are to their surrounding execution environment —that is, if the behavior observed in the system is always the most appropriate according to the surrounding execution environment. Compatibility also

includes the ability of the system to reflect time-based features of adaptations (retract adaptations that are no longer used).

Architectural solutions normally define adaptations of the system as part of its normal evolution process —that is, always moving forward in time. Hence, most adaptations are built on top of previously defined adaptations. As a consequence adaptations persist during the whole life cycle of the system. Nonetheless, behavior of adaptations could be shadowed by new adaptations as a way to retract unused adaptations. Regarding the faithfulness of the solutions to the surrounding execution environment, we note that whenever adaptations are known beforehand, the technique is faithful, however, if adaptations are not known, faithfulness is compromised in order to ensure safety.

Middleware solutions, given their inherent reactivity, are highly compatible with the surrounding execution environment. Moreover, events have an intrinsic notion of temporality, thus adaptations are only available as long as the events they respond to are available.

Language solutions provide adaptations as responses to requests from the surrounding execution environment (e.g., method calls), automatically providing a correspondence with the situation in it. However, adaptations that are no longer in use remain part of the system.

Middleware solutions are appropriate for the development of adaptations compatible with the surrounding execution environment of Dynamically Adaptive Software Systems.

D.5 Extensibility: The extensibility requirement refers to the ability to incorporate adaptations into a software system, even if these are not known beforehand. These new adaptations should not depend on those already defined in the system.

Architectural solutions mainly provide a structure of the system with specific system modules representing the different adaptations. Hence, adaptations are normally known beforehand, and incorporation of new adaptations in the system would have an impact on the architecture of the whole system. Techniques like software upgrades are specifically tailored for the introduction of new adaptations. Similarly, middleware solutions require the modification of the base system in order to respond to new events or adaptations introduced in the system. Likewise, language solutions also need to define all the conditions for which adapted behavior is applicable. Adaptations cannot be incorporated without knowing first the conditions for the adaptation to be applied, and the interaction with other adaptations.

From the three solution categories, none of them seems to fully cover the extensibility requirement. However, specific approaches such as reactive, programming, self-adaptive systems or COP provide support for seamless introduction of adaptations in Dynamically Adaptive Software Systems.

From all surveyed solutions only self-adaptive systems and context-oriented programming support all requirements for Dynamically Adaptive Software Systems. However, as noted in the introduction of self-adaptive systems, these systems comply with the requirements as a whole, but no single implementation complies with all requirements. Context-Oriented Programming implementations comply with all requirements. Hence, this is the chosen solution we use in this dissertation for the development of Dynamically Adaptive Software Systems.

M.1 Interaction: The interaction requirement refers to the ability of defining adaptations to seamlessly interact and be composed with other adaptations defined in the system.

Modeling approaches normally provide a means to specify the interaction between different adaptations. Interaction is often defined by means of dedicated rule systems, where interaction between adaptations is usually described explicitly. This requires to express out *all* possible interactions beforehand.

Formal approaches are conceived as a way to specify the state of the system and its properties. Interaction between adaptations would be expressed by different formulae, requiring to model *every* possible interaction by a formulae, which can become cumbersome for large systems if no automated support to generate such formulae is provided.

Rule-based approaches normally only provide the means for the definition of production rules and the facts to match those rules. However, there is not a defined way in which different production rules interact whenever their facts are matched for a same situation.

Both modeling and state machine approaches are appropriate for the modeling of interaction between adaptations when developing Dynamically Adaptive Software Systems.

M.2 Safety: The safety requirement refers to the ability of software systems to ensure that newly introduced adaptations do not break the behavior already provided by other adaptations.

In order to satisfy this requirement we observed that the majority of surveyed approaches must be extended, allowing the run time introduction of adaptations. Modeling approaches are often static, where all the adaptations are defined during the design of the system.

However, rule-based approaches allow the publication of facts and production rules at any moment in time. Once a production rule is published into the fact space, it is automatically evaluated with all the other production rules, as information is gathered by the system.

Formal approaches usually express the state space of the system by means of state machines, which can introduce new adaptations by means of automata composition (e.g., automata product), and then be verified using

a formal specification of the system properties. In Petri nets too, adaptations can be introduced by means of composition and once composed, all adaptations can be verified by means of property analysis to ensure that the system definition is still correct.

Given the correct extension or composition mechanism, most of the surveyed approaches are thus appropriate to ensure safety of adaptations in Dynamically Adaptive Software Systems.

M.3 Abstraction: The abstraction requirement refers to the ability of the approach to abstract the run-time state of the system in a meaningful fashion. The system should also clearly specify the possible actions in each of the system states, and the interaction between adaptations.

Architectural modeling approaches are abstractions of the system to be used at design time. Normally these abstractions only provide a view of the system structure, but can be complemented (through other model entities) to represent the state and actions of the system. We consider these complements as different models, and hence there is no single representation of totality of the system.

Formal approaches represent the system state by expressing it in terms of formulae or (formal) models. In either case they effectively provide an abstraction of the system states, and the transitions between such states.

Rule-based approaches (partially) abstract the state of the system by the use of facts. The state of the system is guarded within the rete algorithm itself, however, models that explicitly exposed the state of the system. Transitions between states are automatically obtained by the matching of facts in the algorithm.

State machine approaches effectively abstract the states and actions of the system by representing them as a graph-flavored model, where states are nodes of the graph and actions are edges between nodes.

Formal, rule-based, or state machine approaches could all be used as abstraction models of Dynamically Adaptive Software Systems. We decided for Petri nets because, unlike the other approaches, its implicit notions of concurrent states of the model reduces the number of combination of states that need to be represented in the system.

M.4 Decision: The decision requirement refers to the ability of the software system to reason about its run-time properties (be they at run time or at earlier stages of the development cycle).

Architectural modeling approaches are introduced with the sole purpose of providing the means to reason about system properties by means of dedicated analyses.

Formal approaches are by definition reasoning frameworks. The ability to analyze system properties is an inherent property of these approaches, be it by formal verification or theorem proving. It is however unclear,

in some cases, how to capture the run-time properties of the system by means of a formal model. In particular if the model's main focus is not on the run-time behavior of the system.

Rule-based approaches are not concerned with the analysis of system properties, so no decision support for any kind of system property is provided.

State machine approaches provide support for structural system properties, such as composition or hierarchization. Even though they provide only limited support for the analysis of run-time properties of the system, the supported analyses could be used to reason about different properties of the system.

All approaches could be used to analyze different properties about Dynamically Adaptive Software Systems.

From all surveyed approaches, only Petri nets satisfy all the requirements defined for conflict resolution models for Dynamically Adaptive Software Systems and the additional properties presented in Section 3.2.5. Hence, this is the chosen approach we use in this dissertation to model and manage Dynamically Adaptive Software Systems. After having made that choice, the study of how other approaches could be adapted for this purpose was out of the scope of this dissertation.

		Timeliness (D.1)	Granularity (D.2)	Independence (D.3)	Compatibility (D.4)	Extensibility (D.5)	Interaction (M.1)	Safety (M.2)	Abstraction (M.3)	Decision (M.4)
Dynamically Adaptive Software Systems										
Architectural Solutions	Design Patterns	■	■		■		■	■		
	Dynamic Software Upgrades			■		■		■		
	Software Product Lines	■		■				■	■	
Middleware Solutions	Dependency Injection	■			■		■			
	Service-Oriented Architecture	■			■	■			■	
	Event Systems	■			■					
	Self-Adaptive Systems	■	■	■	■	■		■		■
Language Solutions	Metaprogramming	■	■	■			■			■
	Reactive Programming	■			■	■				
	Aspect-Oriented Programming	■	■	■			■		■	■
	Context-Oriented Programming	■	■	■	■	■			■	
Models for Inconsistency Management										
Modeling Approaches	Model Transformations	■	■	■			■		■	■
	Feature-Oriented Domain Analysis	■	■	■	■		■	■	■	■
Formal Approaches	Logic Programming Languages						■		■	■
	Algebras & Logic		■				■	■	■	■
	Model Checking						■	■	■	■
Rule-Based Approaches	Fact Spaces	■		■	■	■		■	■	
State Machine Approaches	Automata	■					■	■	■	■
	Statecharts	■	■	■			■	■	■	■
	Dataflow Graphs	■	■		■	■	■		■	
	Petri nets	■			■	■	■	■	■	■

Table 3.1: Compliance of surveyed approaches with the requirements of Dynamically Adaptive Software Systems.

Chapter 4

Context-Oriented Programming

In this chapter we provide an overview of the COP, a language approach for the implementation of Dynamically Adaptive Software Systems. Systems implemented using COP, usually called context-aware systems, are characterized by the provision of run-time behavior adaptations to the surrounding execution environment of the system. We argue that context aware systems provide the highest dynamicity among the techniques realizing Dynamically Adaptive Software Systems.

The COP paradigm [44] is an emerging programming paradigm for the development of context-aware systems. The COP paradigm introduces dedicated language abstractions to facilitate the definition and modularization of dynamic adaptations in a software system. The motivation behind COP systems is to provide a programming model that embraces pervasive and ubiquitous computing [194], by sensing and effectively using information about the surrounding execution environment of a software system (e.g., location and hardware diagnostic services). The hypothesis of COP is that the environment in which systems execute changes constantly. For example, computer components, and the services they provide appear and disappear constantly as users move around, changing the needs and uses of software systems. Context-aware systems adapt to their surrounding execution environment by means of dedicated behavior (i.e., behavioral adaptations) defined for each of the software services that may profit from it, while keeping users oblivious to the dynamic adaptations of behavior and their interactions.

The overview of the COP paradigm provided in this chapter begins with the description of its characteristic features. Additionally, we evaluate existing COP approaches with respect to support they provide for consistency and predictability. These characteristics are made concrete by means of a concrete implementation of a COP language, Subjective-C [77], which we will later use as our language laboratory for the development of highly dynamic software systems while remaining predictable. This chapter concludes by a summary of the characteristics of COP systems in perspective to the requirements defined

in Section 2.4.

4.1 Introduction to Context-Oriented Programming

Context-Oriented Programming (COP) [44] allows software systems to be easily modularized into *program adaptations* that can be activated and deactivated dynamically at run time. Each adaptation represents a set of particular behaviors, or behavioral adaptations, that depend on specific properties or situations observable in the surrounding execution environment of the system. In the seminal work on COP, Hirschfeld et al. [90] identified four essential properties of COP languages. These languages should provide: (1) the means to specify behavioral adaptations, (2) the means to group adaptations into layers, (3) dynamic activation and deactivation of layers based on context, and (4) the means to explicitly and dynamically control the scope of layers.

Two notions are key for the development of of COP systems, namely *behavioral adaptations* and *contexts*. Before going further we make precise these definitions.

Definition 4.1. *We define a* ***behavioral adaptation*** *as a pice of behavior modifying (replacing or adapting) a particular method defined for the system.*

Different definitions of context have been given in the past [69, 109, 70, 90]. Through this dissertation we use a refined version of context based on the definition originally presented by Dey [53].

Definition 4.2 (**Context**). *A context is an abstraction or a reification of a particular property characterizing a situation in the surrounding execution environment that is semantically relevant for the system.*

Definition 4.3. *An* ***adaptation*** *is composed of two main elements: a particular situation of the surrounding execution environment of the system, a* context*, and a set of* behavioral adaptations *exhibiting specific system behavior associated with that particular context.*

This separation of program adaptations into contexts and behavioral adaptations follows the definition initially given by Loke [127] with respect to the non interchangeability of context situations and activities.

It is worth noting that many COP languages introduce the concept of *layers* [174] as originally proposed by Hirschfeld et al. [90] in order to represent contexts. The difference between contexts and layers is subtle but inessential for our work. Throughout this dissertation we will use the term *context* to denote both contexts and layers without distinction.

The situations that can be represented by contexts can be logical (to the system), physical (to the surrounding environment), endogenous, or exogenous. Examples of contexts are: **(1)** In a mobile application, the battery level property of the device could be used to describe situations as `LowBattery` or

HighBattery (physical, endogenous), **(2)** Web browsers can defined user preferences to determine how to display web pages according to UserPreferences. The browser could adapt, for example, to the UserFontSize or UserLanguage (logical, endogenous), **(3)** In program optimization systems, results of pattern matching algorithms could be used to choose the type of parallelization technique to be used in situations like NRecursiveProcess or I/OHybridLoop (logical, exogenous), and **(4)** In a weather forecast application, the weather information could be displayed based on a service to retrieve the user location represented by geographical situations Brussels or Louvain-la-Neuve (physical, exogenous).

Various COP languages have been proposed. These languages are developed either as extensions of existing languages, or as entirely new languages [4, 160]. Existing COP languages in one way or another all support the four basic properties [90] initially described, based on the underlying technology or implementation techniques used.

From the different existing implementation we noticed that adaptations constitute an important concern of the system's development. Usefulness of dynamic adaptations is proven by their interaction with the base behavior of the system. The dynamics of adaptation are seen in how behavioral adaptations are included to or withdrawn from the system as a consequence of changes in the surrounding execution environment, the extent to which behavioral adaptations impact the system, and how behavioral adaptations are composed. In order to describe the main characteristics of COP systems we broaden the four initial properties taking into account adaptation interaction. A COP language is then characterized by the definition of: **(1)** Modularity of adaptations, **(2)** Selection of adaptations, **(3)** Scoping of adaptations, and **(4)** Composition of adaptations. In addition we discuss an extra dimension about **(5)** Existing support provided in COP languages to manage consistency and predictability of behavior. The following sections describe each of these characteristics and how they are reified in different COP languages. A summary of COP implementations and their properties is provided later in Table 4.1.

4.1.1 Modularity of Adaptations

COP allows expressing behavior that is specific to a particular situation. Hence, any COP language not only must offer the possibility to define multiple behaviors to a given situation in the surrounding execution environment of the system, but it also must be able to define adaptations of the same behavior to different situations in the surrounding execution environment. Adaptations could be structured as independent modules grouping contexts and their associated behavioral adaptations, or as a particular specializations into the respective modules in which the base behavior of the system is defined.

Modularization of adaptations can be divided into two: the way in which adaptations interact with the base system and underlying language, and the way in which adaptations are defined.

COP.1 The *adaptation modularization* property describes the way in which a particular adaptation relates to the base application and to the language underlying the COP abstractions. Two ways of interaction are most prominent in the literature, *class-in-layer* and *layer-in-class* [4].

1. The class-in-layer (CIL) modularization defines adaptations as stand alone modules of the system. Each of such modules constitute the context and all its associated behavioral adaptations. This modularization strategy is good for rapid prototyping, easing the extension of the base application behavior and clean separation of concerns, between the base application and other adaptations. Example implementations of this modularization technique are: ContextL [44], Ambience [74], PyContext [190], ContextS [89], cj [164], ContextJ [6], ContextLua [192], ContextJS [124], SCopJ [98], CoPN [33], PhenomenalGem [148], Flute [9], Context Traits [73].
2. The layer-in-class (LIC) modularization defines behavioral adaptations within the application entity they adapt. Each of the base system modules contains all corresponding behavioral adaptations. This modularization strategy is good for preserving the cohesion of base modules and avoiding scattering of application functionality. Example implementations of this modularization technique are: ContextL [44], ContextLogicAJ [3], ContextR [166], ContextPy [172], cj [164], ContextJ [5], Lambic [185], ContextErlang [71], JCop [6], Subjective-C [77], EventCJ [103], JavaCtx [162], ContextJS [124], CoPN [33].

COP.2 The *definition mechanism* modularization property describes the way and granularity in which adaptations are defined. Adaptations are normally represented as first-class entities of the program.

1. *Layers* define groups of behavioral adaptations related to a same situation in surrounding environment of the system. Layer-based languages are: ContextL [44], PyContext [190], ContextS [89], ContextLogicAJ [3], ContextR [166], ContextPy [172], cj [164], ContextJ [5], JCop [6], ContextLua [192], EventCJ [103], JavaCtx [162], ContextJS [124].
2. *Contexts*, similar to layers, group behavioral adaptations related to a same situation of the surrounding environment of the system. Contexts differ from layers in that they are stateful. That is, contexts have a state that dictate if and how many times the contexts has been activated. Context-based languages are: Ambience [74], Subjective-C [77], JCop [6], SCopJ [98], CoPN [33], PhenomenalGem [148], Context Traits [73].
3. *Predicate methods* are used to capture the global state of the system, not via an explicit program entity, but rather by associating a predicate with a particular behavioral adaptation. Whenever the

predicate becomes true, the method is applied. Lambic [185] and Flute [9] provide support for such technique.

4. Other modularizations techniques similar to those provided by contexts, are defined by the concepts of *modules* and *behaviors*. Behaviors define different behavioral adaptations, and modules group a set of behaviors. These modularization techniques are realized by ContextErlang [71] (as modules and variations), and Flute [9] (as modals and modes).

4.1.2 Selection of Adaptations

COP allows us to adapt application behavior according to specific situations of the surrounding execution environment. Hence, any COP approach must offer the means to dynamically choose the adaptations that are to be executed, such that the behavior accommodates to the circumstances in which the system runs. To this end, the selection of the different behavioral adaptations must be late bound —that is, adaptation of behavior cannot happen *a priori*.[1]

Adaptations are chosen based on changes in the surrounding execution environment of the system. When an adaptation is chosen, its behavioral adaptations are composed with the base system. The process of choosing and composing adaptations is known as **context activation**. That is, the process of selecting an adaptation and composing its behavioral adaptations into the base application, or deselecting an adaptation and withdrawing its behavioral adaptations from the base application. Most COP languages offer specialized language constructs to express activation and deactivation of contexts. In this dissertation we will use the general term of context activation to refer to both introduction and withdrawal of behavior adaptations unless specified otherwise.

Whenever a context activation takes place in the system, four perspectives are taken into account. How does the selection of the adaptation takes place, who is responsible for selecting the adaptation, when is the adaptation selected, and how timely is the adaptation deployed.

COP.3 The *adaptation activation* property describes how a behavioral adaptation comes about in the system. Adaptations are selected in two ways:

1. The *implicit* technique is based on the automatic selection of adaptations as the surrounding execution environment of the system changes —that is, every time there is a change in the surrounding execution environment of the system the behavioral adaptations associated to such situation are made available. Languages presenting an implicit selection of adaptations are: PyContext [190], Lambic [185], EventCJ [103], SCopJ [98], CoPN [33], Flute [9], Context Traits [73].

[1] If software "adaptation" takes place at deployment time, it is called *configuration*, and if it takes place at build time, it is called *customization*. In earlier stages it is called *design*.

2. The *explicit* technique explicitly enables manual selection of adaptations. Usually behavioral adaptations are expressed explicitly as a block of code embedded in the base application (as in layered-based), or by tangling specific language constructs for adaptation activation with the base application (as in context-based languages). The languages presenting an explicit selection of adaptations are: ContextL[44], Ambience [74], PyContext [190], ContextS [89], ContextLogicAJ [3], ContextR [166], ContextPy [172], cj [164], ContextJ [5], ContextErlang [71], JCop [6], Subjective-C [77], ContextLua [192], JavaCtx [162], ContextJS [124], CoPN [33], PhenomenalGem [148].

COP.4 The *activation actor* property describes the actor responsible for the selection of an adaptation. The activation actor is closely related to the adaptation activation property. Three main actors exist in current COP approaches.

1. Adaptation activation is managed by the *client* when the adaptation activation takes place through explicit language constructs at program points specified by the developer. Languages providing a client actor are: ContextL [44], Ambience [74], PyContext [190], ContextS [89], ContextLogicAJ [3], ContextR [166], ContextPy [172], cj [164], ContextJ [5], JCop [6], ContextLua [192], Subjective-C [77], JavaCtx [162], ContextJS [124], PhenomenalGem [148].
2. Adaptation activation is managed by a *service* when adaptation activation is defined based on a generic set of conditions that need to be satisfied in order to provide the behavioral adaptations. For example, by processing external events, or by matching of predicates describing adaptations. Languages providing a service actor are: Lambic [185], JCop [6], EventCJ [103], Flute [9], CoPN [33], Context Traits [73].
3. Adaptation activation is managed by a *context manager* when the responsibility of adaptation activation is left to a third-party entity of the system that verifies if all conditions required for the activation are satisfied. For example, interactions between adaptations are verified whenever they are activated. Languages presenting a context manager actor are: Ambience [74], PyContext [190], ContextErlang [71], Subjective-C [77], SCopJ [98], CoPN [33].

COP.5 The *binding time* property describes the moment in which behavioral adaptations are applied from the set of available adaptations. When an adaptation is selected, the behavioral adaptations associated to the adaptation can be bound at one of three different moments.

1. The *context activation* binding time is used when all of the behavioral adaptations associated with a context are applied as soon as the context is activated —that is, every time the state of a context

changes (from active to inactive or *vice versa*) all behavioral adaptation associated with the context are made available (or unavailable) in the base application. Languages presenting a context activation binding time are: ContextL [44], PyContext [190],ContextS [89], ContextLogicAJ [3], ContextR [166], ContextPy [172], cj [164], ContextJ [5], JCop [6], ContextLua [192], Subjective-C [77], EventCJ [103] JavaCtx [162], SCopJ [98], CoPN [33], Context Traits [73].

2. The *method dispatch* binding time is used when behavioral adaptations are applied as soon as the method they adapt is called. Every time a method is called, the most appropriate behavioral adaptation corresponding to that method is looked up among the available adaptations. Languages providing a method dispatch binding time are: Ambience [74], Lambic [185], ContextErlang [71], ContextJS [124], PhenomenalGem [148].
3. The *reactive method dispatch* binding time is used when behavioral adaptations are looked up among the set of available adaptations. However, whenever a context activation takes place, the behavior currently executed is suspended/paused, and the application continues with the associated behavior appropriate to the newly active context. The new behavior can resume from a previous state or restart the computation. Flute [9] provides a reactive method dispatch binding time.

COP.6 The *adaptation timeliness* property describes how timely the selection of adaptations is. That is, if the behavior of the system exactly represents the situations of the surrounding execution environment, or if it respects the conditions in which method executions started. Three notions of compatibility exist for the timeliness of adaptation selection.

1. The *loyal* method for adaptation selection ensures that the execution of behavioral adaptations completely take place in the same configuration of contexts in which it started. Execution of behavioral adaptations using the loyal adaptation selection may not always represent the situations currently taking place in the surrounding execution environment, but rather the situation in which the observable behavior started. Languages providing a loyal selection are: ContextL [44], ContextS [89], ContextLogicAJ [3], cj [164], ContextJ [5], ContextLua [192], JavaCtx [162], ContextJS [124].
2. The *prompt* method for adaptation selection ensures that behavioral adaptations are made available to the system as soon as their contexts are activated. Languages providing a prompt selection are: Ambience [74], PyContext [190], Lambic [185], ContextErlang [71], ContextJ [6], Subjective-C [77], EventCJ [103], SCopJ [98], CoPN [33], PhenomenalGem [148], Context Traits [73].
3. The *prompt-loyal* method for adaptation selection is a combination

of the loyal and prompt methods. In this method behavioral adaptations currently executing finish in the same configuration of contexts in which they started. However all new method calls take into account the new configuration of contexts. Prompt-loyal selection is introduced in Ambience [74, 31], however Flute [9] also provides this property. In Flute adaptation selection is prompt because the most appropriate behavioral adaptations is used for every method call. Adaptation selection is loyal in the sense that computation can be paused and resumed, providing full compatibility with the surrounding execution environment is guaranteed.

4.1.3 Delimitation of Adaptations

COP allows to define adaptive behavior that only reaches specific parts of the system. Hence any COP approach must provide a means to define the delimitation of adaptations. Scoping of adaptations is important to ensure that adaptations only affect well-defined parts of the program.

Adaptations can be delimited according to three perspectives: the moment in which adaptations are defined, how adaptations are processed, and how adaptations are confined.

COP.7 The *delimitation definition* property describes when the delimitation of adaptations is determined. COP languages provide means for defining the parts of the program for which an adaptation has an effect. Delimitation of adaptations is achieved in two ways.

1. The *static* adaptation delimitation technique explicitly delimits the parts of the application in which the behavioral adaptation defined for a context take place —that is, defined language abstractions clearly delimit the program blocks presenting particular adaptations. Languages presenting a static delimitation definition are: ContextL [44], PyContext [190], ContextS [89], ContextLogicAJ [3], ContextPy [172], cj [164], ContextJ [5], JCop [6], ContextLua [192], JavaCtx [162].
2. The *dynamic* adaptation delimitation technique implicitly uses the set of active contexts to invoke the appropriate behavioral adaptations —that is, behavioral adaptations are available as long as their associated contexts are active. Languages providing a dynamic delimitation definition are: Ambience [74], Lambic [185], ContextErlang [71], EventCJ [103], ContextJS [124], SCopJ [98], CoPN [33], PhenomenalGem [148], Flute [9], Context Traits [73].

COP.8 The *delimitation specificity* property describes the processing unit to which an adaptation is applicable.

1. In a *local* delimitation, behavioral adaptations are only available in the thread in which the activation of the associated context occurred.

Languages providing a local delimitation are: ContextL [44], PyContext [190], ContextS [89], ContextR [166], ContextPy [172], ContextJ [5], JCop [6], ContextLua [192], Subjective-C [77], JavaCtx [162], ContextJS [124], CoPN [33].

2. In a *global* delimitation, behavioral adaptations are made available for all the running threads in the system. Languages providing a global delimitation are: Ambience [74], ContextLogicAJ [3], cj [164], Lambic [185], ContextErlang [71], Subjective-C [77], EventCJ [103], ContextJS [124], SCopJ [98], CoPN [33], PhenomentalGem [148], Flute [9], Context Traits [73].

COP.9 The *delimitation confinement* property describes the unit in the system which adaptations are applied to. Adaptations can be delimited to particular object instances or to complete families of objects.

1. In *per class* adaptation scoping, behavioral adaptations are effective for all instances of the class in which the context activation occurs. Languages providing per class confinement are: ContextL [44], Ambience [74], PyContext [190], ContextS [89], ContextLogicAJ [3], ContextR [172], cj [164], ContextJ [5], Lambic [185], ContextErlang [71], JCop [6], ContextLua [192], Subjective-C [77], JavaCtx [162], SCopJ [98], CoPN [33], Flute [9].
2. In *per instance* adaptation scoping, behavioral adaptations are effective only for a particular object instance. Languages providing per instance confinement are: EventCJ [103], ContextJS [124], Context Traits [73].

4.1.4 Composition of Adaptations

COP allows the adaptation of systems to their surrounding execution environment by allowing selection and delimitation of adaptations. However, any such technique would be ineffective if adaptations would be too burdensome to produce. Hence any COP approach must provide the means to define new adaptations as increments of previous ones. By combining adaptations in flexible ways, it is easier to obtain different useful behaviors that can then be chosen flexibly.

Multiple contexts can be simultaneously active in the surrounding execution environment. In order to provide new and useful behavior from these contexts, it is possible to combine them by means of reuse. Similarly to stratification of objects into object hierarchies or combination qualifiers [21], COP needs to provide a way to define which behavioral adaptations reuse behavior from which others.

Adaptations can be composed based on two mechanisms: ordering of behavioral adaptations through disambiguation, and the declaration of interaction between adaptations.

COP.10 The *disambiguation* composition property describes a common way of combining different pieces of behavior by ordering them. The ordering in which behavior adaptations are selected is usually provided beforehand by the user or the developer, but can also be decided upon at run time. Defining a simple ordering of behavior adaptations is fairly rigid, since orderings need to be anticipated and have a coarse level of granularity (e.g., classes or behavioral modules). More dynamic possibilities of orderings include the following:

1. The *activation order* technique defines the order of behavioral adaptations according to a *timestamp* property. That is, whenever adaptations are selected the activation is annotated with a timestamp. When a method is called the behavioral adaptations associated with more recently activated contexts will be applied first. This ordering technique is particularly flexible, as the observed behavior may change by changing the order in which variations are selected. Languages providing activation order disambiguation are: Subjective-C [77], CoPN [33], PhenomentalGem [148], Context Traits [73].
2. The *explicit priorities* technique orders behavioral adaptations by annotating them with a *priority* value. This priority defines the sequence in which adaptations are combined. For example, numerical priorities from greatest to smallest, or by the ordering of combination qualifiers [21]. Definition of priorities may yield a rigid combination technique of adaptations. However, at such a fine level of granularity, behavioral adaptations can be combined according to the particular needs of every method. Languages providing priorities disambiguation are: ContextL [44, 49], ContextS [89], ContextLogicAJ [3], Subjective-C [77, 123], CoPN [33].
3. The *selection* technique dynamically orders behavioral adaptations by exploiting an activation order-like strategy. In a selection strategy behavioral adaptations are combined in a stack-like structure. The observed behavior of the application is that of the adaptation closer to the top of the stack (i.e., the one that has been selected most recently). Languages providing selection disambiguation are: ContextlogicAJ [3], ContextR [166], ContextPy [172], ContextJ [5], ContextErlang [71], ContextLua [192], EventCJ [103], SCopJ [98], PhenomenalGem [148].
4. The *declaration order* technique orders behavioral adaptations according to the lexical order in which they were defined in the application. The definition can be either the definition of the first-class entity itself (e.g., context), or the definition of the adaptations associated to that entity. Whichever the case, at run time the behavior is combined by choosing the first adaptation defined in the application. Languages providing a hierarchical disambiguation are: ContextL [44], Ambience [74], PyContext [190], ContextS [89], Con-

textlogicAJ [3], cj [164], ContextJ [5], ContextErlang [71], JCop [6], JavaCtx [162], ContextJS [98].

COP.11 The *adaptation interaction* composition property describes the way in which adaptations may relate to each other. Relations can be structural or logical. A structural relation is given by the structure of system entities and modules for which adaptations are defined. A logical relation is given by a set of constraints an adaptation must satisfy with respect to one another. Such relations can be used at run time to influence the way in which adaptations are composed.

1. The *delegation* interaction between adaptations is implemented by allowing to reuse the behavioral adaptations provided other adaptations. Whenever an adaptation is defined as an increment of an existing one, a delegation relation is set between them. In such a scenario, the later adaptation can override the behavior of former adaptations, or it can increment it. In case the behavior provided by the former adaptation is needed, it can be accessed by means of a dedicated language constructs. This technique of adaptation composition resembles that of code reuse provided by means of delegation in *prototype-based* languages, or inheritance in *class-based* languages. Languages providing a delegation interaction are: ContextL [44], Ambience [74].
2. *Dependencies* are used to define interactions among adaptations. Such interactions determine how adaptations are selected. Dependencies can be expressed by means of rules or transformations specifying how adaptations interact. Normally, dependencies define the conditions under which a group of adaptations is combined for a given change in the surrounding execution environment. For example, allowing or denying adaptations activations based on the state of other adaptations. Languages providing dependencies interaction are: ContextL [44], Subjective-C [77], EventCJ [103], ContextJS [124], SCopJ [98], CoPN [33], PhenomenalGem [148], Context Traits [73].
3. *ADT* can be introduced to describe the way in which adaptations interact with each other. The Abstract Data Type (ADT) encapsulates the adaptations that can be made available, organizing explicitly all allowed combinations. For example by stating that at least one of two adaptations must be available at all times. ContextErlang [71, 163] provides an ADT combination of adaptations.
4. A *state change configuration* is used to express the way in which adaptations interact with the surrounding execution environment rather than with each other. Such a configuration definitions when adaptations are suspended, resume, restarted or stopped. Flute [9] provides a state change configuration of adaptations.

Table 4.1 summarizes currently existing COP languages and their differences with respect to the four characteristics of adaptations described in this section. The languages are ordered chronologically by first date of publication.

Note from Table 4.1 most COP languages do not provide any support to manage the interaction between adaptations (the adaptation interaction property **COP.11** consisting of delegation, dependencies, ADT, and change configuration). This lack of support for the interaction between adaptations partially motivates the approach taken in this dissertation for the study of Context-Oriented Programming, and Dynamically Adaptive Software Systems in general, from an interaction management view point. In Section 4.2 we provide a more in-depth outlook of the existing support to manage a consistent interaction between adaptations in COP languages.

4.2 Consistency Management in COP Languages

Orthogonally to the characteristics of COP systems given in the previous section, we identify an additional characteristic: the consistency and predictability of the observable behavior of the system. Recent advance in different COP languages have identified the importance of maintaining a consistent execution of the system functionality in the presence or absence of behavioral adaptations during system execution. The interest in system predictability arises from the observation that interaction between adaptations and the base application is not always described, as it is possible to see form property COP.11 in Table 4.1. As systems grow and more complex it becomes harder to foresee all such interactions, possibly leading to conflicting or contradicting behavioral adaptations being simultaneously available in the system. We refer to these situations as **behavioral inconsistencies**.

Remember from Section 2.3.3 that already in the small example of the maps application it is possible to spot situations in which behavioral inconsistencies could occur. These situations serve as inspiration to define three sources of behavioral inconsistencies. This list is not meant to cover all possible cases for all different language features from all existing COP languages, but it aims to select situations that are common to all.

Dynamicity: Adaptations are introduced to and withdrawn from the system arbitrarily over time as a consequence of changes in the surrounding execution environment of the system. In the maps application the dynamicity property can be seen in the activation of the `WIFI` context. This context can be activated intermittently as the user roams around and wireless networks are joined and left behind, making its provided services constantly available and unavailable. Behavioral inconsistencies may arise due to dynamicity whenever behavioral adaptations occur when they should not, for example, if the location of the user is requested for the `POSITIONING` adaptation whenever there is no service (`NLBS` or `GPSANTENNA`) providing

[2] `http://www.p-cos.net/context-scheme.html`

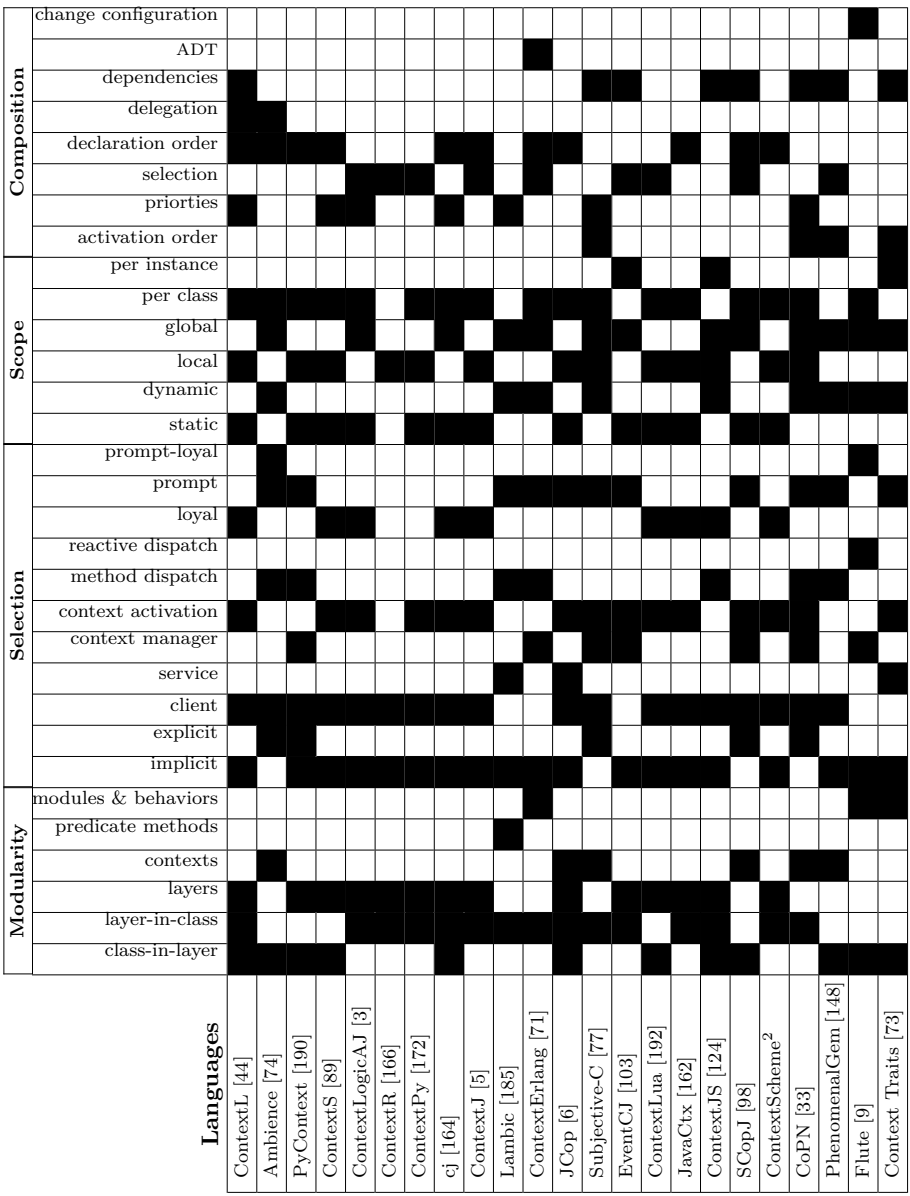


Table 4.1: Properties of COP approaches.

such method, or when they do not occur when they should, for example not enabling the NLBS service when there is a WIFI connection. We refer to these situations as the **adaptation fragility problem**.

Interaction: Multiple adaptations can be active simultaneously, thus their behavioral adaptations may be accidentally rendered incorrect by the presence of other available behavioral adaptations. As an example of such undesired interaction, consider the activation of the POSITIONING service (by the availability of the GPSANTENNA) which broadcasts the position of users, and the activation of the PRIVATE context (by manual configuration of the user) which conceals all user information in the maps application. When the two contexts are active simultaneously, their behavioral adaptations are in conflict.

Multiplicity: Situations in which an adaptation may become available can arise in different ways, for example via their interaction. Hence, context objects can be activated more than once. Semantics of context activations should consider the multiplicity of context activations into account to ensure that behavioral adaptations associated with contexts are available as long as they are needed (i.e., by some other adaptation or are available in the surrounding execution environment). As an example of multiplicity, consider the POSITIONING context which can be activated four times: it can be directly activated, or indirectly by the activation of the GPSANTENNA, GSMLOCATION and NLBS contexts. Disconnection of any of these services should not immediately disable availability of the POSITIONING service as long as it is still needed to be active via other context. For example, if only the NLBS service disappears, POSITIONING can still happen through one (or a combination of) the other services.

Different proposals have been made to ensure a consistent and predictable behavior of COP systems. Unfortunately most of these proposals are language-specific and no general consensus has been reached regarding which situations impact all context approaches and which ones occur due to design choices made in a specific language.

An initial proposal to manage consistency of COP systems was to incorporate contexts in the software design process by means of CODA [49]. CODA considers software systems as an aggregation of features describing the application behavior, which can be adapted according to the context in which the system is used. CODA defines a set of relationships that can be declared between adaptations in order to deal with the problem of unanticipated adaptation interaction. These relationships extend the existing relationships (e.g., and, optional, or mandatory) of feature diagrams [106], for example, by means of inclusion and exclusion relations between adaptations expressing respectively that adaptations must co-exist with each other, and adaptations cannot co-exist with each other. Additionally, CODA implements different resolution strategies to order behavioral adaptation in case of adaptations interactions. The work on CODA

is extended in two directions. On the one hand ContextL introduced declarative layer constraints [43] and the use of the reflective meta layer of ContextL to constrain activation of contexts [48, 45]. On the other hand, based on the interaction resolution strategies, a set of policy rules can be implemented and verified when contexts are activated [51, 50].

An example of a CODA diagram and its resolution strategies for the maps application Section 2.3.3 is given in Figure 4.1. In the figure rounded squares represent the *varaition points* of the system, and squares are their available adaptations. In case multiple adaptations are enabled simultaneously, as it is the case for the `GPSAntenna`, `GSMLocation`, and `NLBS` adaptations, a priority is given to each of them. Priorities are represented by a annotating the relationship between adaptations with a number, as shown in the figure.

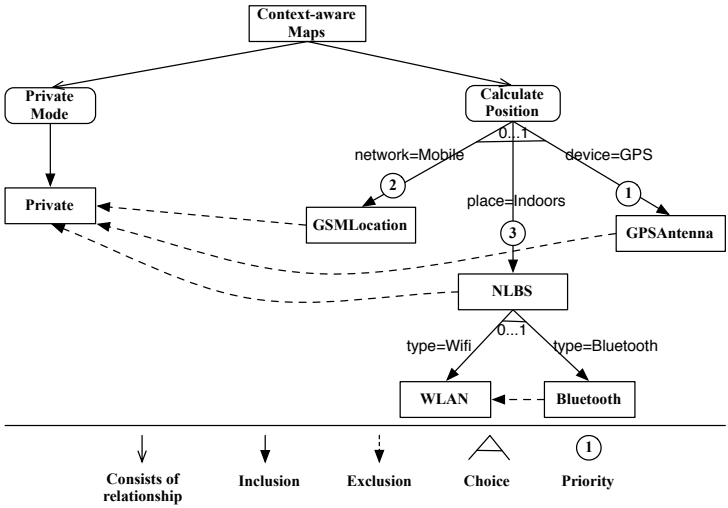


Figure 4.1: CODA diagram for the maps application.

The approaches introduced in ContextL to ensure consistency have an important drawback. As systems grow it becomes harder to see interaction between adaptations. Moreover, the diagram and the required knowledge behind it could get cluttered. Introduction of new interactions between adaptations requires the definition of new notations. In addition, due to the difference between the high-level definition of relationships in the diagram and their implementation, interaction between adaptations could be missed raising inconsistencies at run time.

Similarly to the verification of policy rules introduced in ContextL, external rule engines could be used for the verification of specified rules, as explained in Section 3.2.4. In particular forward chaining engines such as Crime [136], could be used for the evaluation of predicates associated with context activation. Such predicates could be used (but are not currently in use) in languages that explicitly exploit context changes by means of predicates such as Lambic [185]

or Flute [11]. Flute provides instead a declarative way to describe the different situations that could take place when a context change occurs while a behavioral adaptation is being executed. The programming model used in Flute is based on the concept of continuations [85]. Following the ideas of continuations, activation constraints that can be declared in a context refer to whether the process should suspend, abort, restart or resume the execution of a behavioral adaptation.

EventCJ also allows the definition of context transition rules [103] which, similarly to policy rules in ContextL, allow to change the set of active contexts (of an object instance in EventCJ) as a consequence of an event associated with contextual information. Transition rules can enforce constraints for the activation of different contexts. Context transition rules in EventCJ are modeled with a finite automaton which allows to verify safety properties of context transition rules by means of model checking techniques.

Snippet 4.1 shows an example of the definition of events, and the declaration of transition rules between adaptations for some of the adaptations of the maps application introduced in Section 2.3.3. Events are triggered by particular situations in the surrounding execution environment of the system, for example, the `NLBSEvent` defined in Line 7 is triggered whenever a connection is detected. Triggering of events activates events according to the definitions given for each event (Lines 13 through 19). The first transition rules expresses that if the `PRIVATE` adaptation is not active, then it is possible to activate the `POSITIONINGEVENT`. The second rule expresses that the `NLBS` adaptation can be activated if the `GPSANTENNA` or `GSMLOCATION` adaptations are not active.

```
1  direction CalculatePosition {
2    declare event PositioningEvent(Object o, int s)
3       :after call(void onEventReceived(s))
4      &&target(o)&&args(s)
5      &&if(s==BROADCASTING)
6      :sendTo(o);
7    declare event NLBSEvent(Object o, int s)
8      :after call(void onEventReceived(s))
9      &&target(o)&&args(s)
10     &&if(s == CONNECTED)
11     :sendTo(o);
12   //Transition rules
13   transition PositioningEvent:
14     not PRIVATE activate POSITIONING;
15   transition NLBSEvent:
16     not GPSANTENNA activate NLBS |
17     not GSMLOCATION activate NLBS;
18   transition ConnectivityEvent:
19     BLUETHOOTH switchTo WLAN;
20 }
```

Snippet 4.1: Event declarations and layer transition rules for the maps application in EventCJ.

Having defined the transition rules, EventCJ allows to automate the transition of such rules into Promela [93], a language for the specification of processes. Such a specification is used for the verification of consistency properties about the layer transition rules. Snippet 4.2 shows an example of the transition rules

specification using Promela. The first process (Lines 1 through 16) describes the transition rules for each event. Events are gathered via a `channel|` variable. The second process (Lines 17 through 25) describes the environment of the base application. In this example description `PrivateEvent` can be triggered at any time, `PositioningEvent` can only be triggered if the `GSMLocationEvent` event was triggered before.

```
1  active proctype Object() {
2    do
3    :: channel ? NLBSEvent ->
4    S0: if
5      :: (GPSAntena==Inactive) ->
6        atomic {NLBS=Active}
7      :: (GSMLocation==Inactive) ->
8        atomic {NLBS=Active}
9      fi
10   :: channel ? ConnectivityEvent ->
11   S1: if
12     :: (Bluetooth==Active) ->
13       atomic {Bluetooth=Inactive; NLBS=Active}
14     fi
15   od
16 }
17 active proctype Env() {
18   do
19   :: channel ! PrivateEvent
20   :: channel !  GSMLocationEvent ->
21     do
22     :: channel ! PositioningEvent
23     od
24   od
25 }
```

Snippet 4.2: Layer transition rules translated to Promela.

Once the system specification has been generated, it is necessary to write the properties to be verified by the model checker. The Linear Temporal Logic (LTL) formulae shown in Snippet 4.3 express the conditions that should be satisfied in the system. The `[]`, `V`, and `U` denote the temporal operators *globally*, *release* and *until*, respectively.

```
1  [] ! (GPSAntenna && GSMLocation && NLBS)
2  [] ! (Bluetooth && WLAN)
3  [] ! (Private && Positioning)
4  [] (Positioning U (GPSAntenna || GSMLocation || NLBS))
```

Snippet 4.3: System properties specification in LTL.

The adaptation transition rules provided in EventCJ are a step forward from those initially presented in ContextL because the safety of rules can be verified beforehand. However, transition rules still have to be specified as part of the EventCJ program, and the properties transition rules should satisfy as LTL formulae (for their verification using a model checker, such as SPIN [93]). This requirement of a multiple specification of the system makes its development cumbersome and error prone, as there could be inconsistencies between the specifications. Additionally, note that EventCJ has a problem coping with the multiplicity of behavioral adaptations. The automaton model of transition rules only allows to deal with binary states of contexts, thus adaptations that interact

with each other cannot be active simultaneously in the system. Composite layers have been introduced in EventCJ to deal with adaptation interactions [104], however, this approach requires the definition of new adaptations making up the interaction.

A recent proposal based on context ADTs is used in ContextErlang [71, 163] to manage contexts. Unlike previous approaches, ADTs do not deal with contexts independently, rather ADTs gather sets of valid adaptation combinations. Contexts that interact with each other are encapsulated into an ADT which allows to easily check if such interactions are safe. ADTs are represented as a fixed size stack structure, where each slot in the stack can have one of three types: *adaptable*, *switch*, or *free*. Each slot specifies which activations are valid for a group of adaptations. An example of the ADT syntax and a concrete example using the maps application are shown in Snippet 4.4. In Snippet 4.4(b) the activation of the `WLAN` adaptation is left as a `free_slot` (Line 1), the `PRIVATE` and `POSITIONING` adaptations are defined so that only one of them can be active at a time defined as a switch slot (Line 2), and the `GPSANTENNA`, `NLBS`, `GSMLOCATION` adaptations are defined as activatable slots that can be active or not (Line 3). The las two lines in Snippet 4.4(b) are used to create and start the corresponding Erlang agents.

```
CONTEXT_SPEC ::= [ SLOT_SPEC* ]
SLOT_SPEC ::= { Slotname, SLOT }
SLOT ::= SWITCH_SLOT
       | ACTIVATABLE_SLOT
       | FREE_SLOT
SWITCH_SLOT ::= [(Varname1,)*
      {Varname2, activate}
      (, Varname3)*]
ACTIVATABLE_SLOT ::= {Varname}
       | {Varname,active}
FREE_SLOT ::= free_slot
```

(a)

```
1  Spec = [{wlan, free_slot},
2       {broadcasting, [{PRIVATE, active}↩
            , POSITIONING]},
3       {connectivity, {BLUETOOTH, active↩
            }}},
4       {positioning, {GPSANTENNA, NLBS, ↩
            GSMLOCATION}}],
5
6
7  Context = context_ADT:create(Spec),
8  user:start_link(AgentId, Context)
```

(b)

Snippet 4.4: (a) Syntax specification of the ContextErlang ADT [163]. (b) Context ADT for the web booking application.

The use of ADTs for the consistency management of adaptation activation presents a fresh look with respect to previously proposed approaches. However, this approach evidences the same problems as those proposed before. That is, the definition of the ADT and the state of its slots needs to be known beforehand by programmers, even more, currently there is no automated way to verify the validity of defined rules. Additionally, whenever adaptations are tightly coupled, their interactions will become part of one single ADT, which boils down to having a set of transition rules.

González et al. [77] propose context dependency relations in Subjective-C to define interaction between contexts. Context dependency relations take inspiration from those defined in CODA and the association of context activation

events with specific constraints defined by the dependency relations. Context dependency relations are implemented in Subjective-C, which Section 4.3.3 explains in depth.

Finally, Context Traits [73] offer an approach oriented to context composition. In Context Traits, traits encapsulate coherent sets of methods. Each trait can be seen as a specification of methods appropriate to execute in a particular context of the surrounding execution environment. Objects are then realized as composition of traits. However, different traits may provide definition for the same behavior (i.e., they provide behavioral adaptations of the same method) yielding a conflict in the composition, which will be the method used at run time? To solve this problem Context Traits allows the definition of composition policies. Objects are composed by means of a composition policy $P(S)$. Multiple policies can be defined in the system. Composition policies are functions that, given a set of traits S, provide a trait without method conflicts. Each resolution trait is unequivocally defined by its composition policy $P(S)$. In case no composition policy is defined by programmers, Context Traits provide a default composition policy that consists in the activation time of each trait in a set S of traits —that is, the default composition policy corresponds to the activation order disambiguation composition property of COP systems (COP.10).

			Problems			
Approach	**Language**	**Mechanism**	method fragility	multiplicity	interaction	automated verification
Design process	ContextL	CODA relationships	■			■
	Subjective-C	Context graph model		■	■	
Reflective layer	ContextL	Run-time interaction constraints	■			■
Rule engines	Lambic	Forward selection of adaptations	■			■
Transition rules	ContextL EventCJ	Satisfiability of predicates. Rules defined per context combination	■			
			■		■	
ADT	ContextErlang	Clustering of interaction			■	
State change configuration	Flute	Independently defined per context. Continuation rules	■			■
Context dependency relations	Subjective-C	Run-time interaction constraints		■	■	
Composition Policies	Context Traits	Composition policies rules $P(S)$ defined per context combination		■	■	

Table 4.2: Consistency approaches in COP systems.

Table 4.2 shows a summary of the existing approaches for the consistency management in COP languages, and the problems not yet tackled by each of the approaches. It is apparent from the table that the recurrent problems through the approaches for consistency management in COP systems are verification,

support for multiplicity of contexts, and the fragile method problem. We argue that a programming model for the consistency management of COP systems must address all of this problems. Chapters 6 and 7 discuss how the basis proposed for the development of consistent Dynamically Adaptive Software Systems proposed in this dissertation addresses such problems.

4.3 Context-Oriented Programming in Subjective-C

Subjective-C is a language extension of the Objective-C language[3] that enables COP. Subjective-C was designed with the purpose of enabling the development of context-aware applications for mobile devices [77]. Subjective-C is a full COP language, which means that it provides all the functionality for the definition of modular adaptations, their selection, scoping, and composition. In the following we describe which of the properties for these concepts are implemented in Subjective-C.

4.3.1 Context Objects: Modularity of Adaptations

In Subjective-C adaptations are modularized by means of *context objects*. A context object, or context for short, is a first-class program entity defined as an abstraction of the particular situation in which the system executes. Contexts normally provide a semantically meaningful definition of a situation in the surrounding execution environment of the system, for which specific behavior should be provided. For example, having a phone charge of 150mAh, is represented as a `LowBattery` context for a particular system, or a positioning coordinate $50°50'N\,4°21'E$, represented as the `Brussels` context.

In Subjective-C contexts are chosen as the definition mechanism for the modularity property (**COP.2**). Using contexts behavioral adaptations can be introduced using a dedicated language construct, `@context`(*context-name*). This construct takes the name of the context as a parameter and returns the corresponding context object. Snippet 4.5 shows as example the definition of the `WLAN` context described for the maps application (Section 2.3.3).

```
SCContext *wlan = @context(WLAN);
```

Snippet 4.5: Subjective-C definition of a `WLAN` context

Context objects reify situations of the surrounding execution environment by associating behavioral adaptations to them.[4] In Subjective-C behavioral adaptations are essentially regular Objective-C methods with a special `@contexts` annotation, followed by the name(s) of the context(s) associated with such behavior. Thus, Subjective-C provides a layer-in-class adaptation modularization

[3] `https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html`

[4] In the literature behavioral adaptations are sometimes referred to as partial behavior definitions, behavior variations, or context-dependent methods.

property (**COP.1**). Snippet 4.6 shows a behavior adaptation defined for the method `broadcastPosition`, which is associated with the `WLAN` context.

```
@contexts WLAN
-(CLLocation *) broadcastPosition {
  //Get the position from the available service
   self.pos = [WLANConnector calculatePosition];
   return pos;
}
```

Snippet 4.6: Behavioral adaptation definition for the `WLAN` context in Subjective-C.

In most cases behavioral adaptations do not re-implement the base behavior of the system in its entirety, but only require to adapt part of the behavior and reuse the previously defined behavior. To access such existing behavior Subjective-C defines the `@resend()` language construct. A `@resend()` invocation works much like a *super call* in object-orientated languages. Whenever a `@resend()` is called, the next applicable method for the current message call (the exact meaning of this "next applicable method" is explained in Section 4.3.3). An example of the use of `@resend()` is shown in Snippet 4.7 for the maps application. Here the call to `@resend()` of Line 17 will call the method defined in Line 2.

```
1  @contexts Positioning
2  -(CLLocation *) broadcastPosition {
3     //Draws the calculated position in the map
4     if( pos == nil)
5       pos = [self dummyPosition];
6     return pos;
7  }
8
9  @contexts NLBS
10 -(CLLocation *) broadcastPosition {
11   //Get the position from the available service
12   CLLocation *temppos = [[NLBS service] calculatePosition];
13   if(self.pos)
14     self.pos = [self refinePositionWith: temppos];
15   else
16     self.pos = temppos;
17   return @resend();
18 }
```

Snippet 4.7: Method resend in Subjective-C.

In Subjective-C adaptations are composed of a context and a list of the behavioral adaptations associated with it.

4.3.2 Context Activation: Selection and Scoping of Adaptations

In Subjective-C adaptation selection takes place based on the state of the adaptation's context object. A context object can have one of two states, active or inactive. A context is said to be **active** if its defining situation is part of the

surrounding execution environment. A context is said to be **inactive**, if the situation for which it is defined is not longer part of the surrounding execution environment.

In Subjective-C adaptations are confined using the per class delimitation confinement property (**COP.9**). that is, whenever a context becomes active (i.e., an adaptation is selected) its associated behavioral adaptations need to be composed with the running system. Similarly, if a context becomes inactive (i.e., an adaptation becomes deselected) its associated behavioral adaptations need to be withdrawn from the running system.

For example, in the maps application, whenever the device detects a wireless local area network to which it can connect, the corresponding `WLAN` context is made active. Similarly, if the device moves to a place where there are no detected `WLAN`'s, the context is made inactive. Subjective-C implements an explicit adaptation activation property (**COP.3**), where context activations and deactivations are signaled explicitly by means of the `@activate`(*context-name*) and `@deactivate`(*context-name*) constructs, respectively. These constructs are used directly by the programmers within the base application, adaptation activations are managed by the client (**COP.4**). Snippet 4.8 shows how these constructs are used in the case of the `WLAN` context.

```
//If Wifi connection is detected
@activate(WLAN);
//If no Wifi connection is detected
@deactivate(WLAN);
```

Snippet 4.8: Activation and deactivation of context `WLAN` in Subjective-C.

Context activations can also be managed by a *context manager* (cf. Section 4.4.1) activation actor (**COP.4**) that tracks the set of active contexts in the system by using *activation counters* [74]. Activation counters work similarly to the retain/release logic of memory management systems based on reference counting. Every time a context is activated by means of an `@activate()` message, its activation counter is incremented by 1. If the context is sent a `@deactivate()` message, then its activation counter is decremented by 1 (if the counter is already zero, the activation counter is not decremented). The set of active contexts in the system is composed of the contexts for which their activation counter is greater than zero; the set of inactive contexts is composed of the contexts for which their activation counter is exactly zero.

Behavioral adaptations are selected using the context activation binding time property (**COP.5**). That is, whenever a context becomes active, all of its associated behavioral adaptations are made available to the system. For example, when context `NLBS` is activated, its associated `broadcastPosition` method is made available to the system —that is, it will be the method called when requesting the position of the user. Whenever the `NLBS` context is deactivated, its associated behavioral adaptations are made unavailable to the system. In particular the `broadcastPosition` method defined for the `NLBS` context will no longer be called.

Adaptations are scoped twofold by taking into account the set of active contexts. First, since the set of active contexts is unique and accessible to the whole system whenever a context is active, its associated behavioral adaptations are made available to all executing threads in the system. Such behavioral adaptations are globally scoped. Second, the set of active contexts is also used to determine the correct behavior to be selected whenever a method is called. Method calls are always looked up first among the behavioral adaptations provided by active contexts (e.g., in the current context). If there are behavioral adaptations for the method, these are called, otherwise the base method behavior is called. This implementation of the introduction and withdrawal of adaptations accounts for a prompt adaptation timeliness property (**COP.6**) and a dynamic delimitation definition property (**COP.7**). In addition to this basic behavior for global context scoping, Subjective-C recently introduced an additional scoping mechanism that allows the scoping of contexts local to a particular thread of execution. Subjective-C enables both local and global delimitation properties (**COP.8**). However, this extension requires a modification of the context representation, selection and scoping mechanisms. We further discuss in Section 9.2 how local context scoping works in Subjective-C.

4.3.3 Context Interaction: Composition of Adaptations

In this section we discuss the last piece of the COP characteristics in Subjective-C, how adaptations compose, and interact with each other. Three methods are provided for the composition of contexts and their interaction. The first method, **context dependency relations** is used to define interaction between contexts, enabling their composition. Context dependency relations describe which contexts may be used in combination with other contexts, and which contexts should not, following the dependencies adaptation interaction property (**COP.11**). The other two composition methods, *context activation time* and *method priorities* which correspond to the disambiguation property (**COP.10**) and are used to define how behavioral adaptations are composed at run time.

When a context activation takes place, context dependency relations define the restrictions and effects of such activation with respect to other contexts defined in the system. In the general case, a context dependency relation between two contexts `A` and `B` imposes conditions on the activation and deactivation of both contexts with respect to each other. However, not all context dependency relations necessarily define conditions for both activation and deactivation of the two contexts.

Context objects are composed in the system by means of context dependency relations in a so called **context dependency graph** data structure. The nodes in the graph are connected to each other through edges expressing the semantics of the context dependency relation between two nodes. If no direct context dependency relation exists, the context objects may still be indirectly related, for example, via a third context which has a context dependency relations with both.

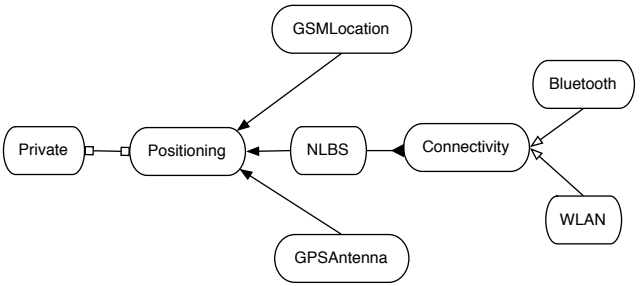


Figure 4.3: Context dependency graph for the maps application.

Figure 4.3 shows an example of the **context dependency graph** for the maps application. Each of the named nodes in Figure 4.3 represents a context, and the decorated edges represent the 4 existing dependency relations in Subjective-C [77]. These context dependency relations are:

Exclusion represents the situation in which two contexts cannot be active at the same time. That is, activation of a context is disallowed if its excluding context is already active. However, both contexts may be simultaneously inactive. An exclusion dependency relation between two contexts is represented graphically by edges with empty squares at both ends (□–□), as is for instance the case for the PRIVATE and POSITIONING contexts in Figure 4.3.

Weak inclusion represents the situation in which the activation and deactivation of the source context automatically triggers the respective activation and deactivation of the target context. However, the dependency is weak in the sense that the target context can still be activated or deactivated independently of the source context. A weak inclusion dependency relation is represented graphically by edges ending with empty triangles (–▷), as for instance the case for the WLAN (source) context which weakly includes the CONNECTIVITY (target) context (WLAN–▷CONNECTIVITY) in Figure 4.3.

Strong inclusion represents the situation in which, similarly to a weak inclusion, the activation (deactivation) of the source context automatically triggers the activation (deactivation) of the target context. In this case however, the inclusion is said to be strong because the deactivation of the target context automatically triggers the deactivation of the source context. However, the target context can still be activated independently of the source context. A strong inclusion dependency relation is represented graphically by edges ending with full triangles (–▶), as is for instance the case for the NLBS (source) context which strongly includes the POSITIONING (target) context (NLBS–▶POSITIONING) in Figure 4.3.

Requirement represents the situation in which the activation of the source context is possible only if the target context is already active. This restriction implies that if the target context is inactive, the source context must necessarily be inactive. A requirement dependency relation is represented graphically by edges ending with inverse full triangles (–◀), as is for instance the case for the NLBS (source) which requires the CONNECTIVITY (target) context (NLBS–◀CONNECTIVITY) in Figure 4.3.

When a context is to be activated or deactivated, a request is sent to each related context in order to check its state. Based on the constraints imposed by the dependency relation between contexts, the (de)activation request is accepted or not. Note that, since activation of a context may trigger that of another context, every request made to a context must be forwarded to all other contexts with which it has a context dependency relation.

Example 4.1. As an example of the interaction of contexts under the influence of context dependency relations we explain the process different contexts in the maps application with a context configuration as shown in Figure 4.3. Suppose that no context is active in the application, and there is a request to activate context WLAN by a call to @activate(WLAN). Context WLAN has only one related context in the context dependency graph, context CONNECTIVITY. To activate WLAN we must activate CONNECTIVITY. CONNECTIVITY has three context dependency relations, two weak inclusion dependencies with BLUETOOTH and WLAN as the target, and a requirement dependency with CONNECTIVITY as the target. None of these relations impose constraints on the activation of the target context, nor does the activation of the target lead to any consequences. Context CONNECTIVITY is then activated. As this is the only dependent context for WLAN, can then also be activated.

Suppose now that the NLBS context is requested for activation by calling @activate(NLBS). Context NLBS has two context dependency relations, a requirement dependency with CONNECTIVITY as the source, and a strong inclusion dependency with POSITIONING for which it is the source. According to the constraints imposed by the requirement dependency the source context can be activated only if the target context is. A message is sent to the target context to verify if it is active or not, since the CONNECTIVITY context was previously activated, NLBS could be activated. The strong inclusion dependency imposes the constraint that every activation of the source must activate the target. Hence an activation message is sent to the POSITIONING context. The POSITIONING contexts has four context dependency relations, three strong inclusion dependencies with contexts NLBS, GSMLOCATION, and GPSANTENNA for which it is the target, and an exclusion dependency with the PRIVATE context. The strong inclusion dependencies do not impose constraints on the activation of the target context. The exclusion dependency constraints the contexts where at most one of them can be active. Hence a message is sent to the PRIVATE context to verify whether it is active or not. Since the PRIVATE context is not active, the POSITIONING context can be activated, this means that the NLBS context can be activated.

After these two activation requests, the active contexts in the system are `WLAN`, `Connectivity`, `NLBS`, and `Positioning`. All other contexts remain inactive.

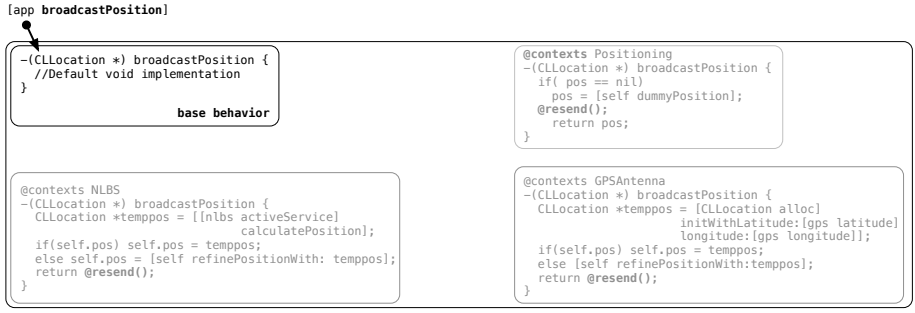
Note from Example 4.1 that forwarding activation requests may be a costly operation at run time when contexts graphs are fully connected (i.e., there is a path between every two nodes of the graph). In a fully connected context graph every activation request is forwarded to all contexts available in the system.

Different active contexts may however define behavioral adaptations for the same method. Such behavioral adaptation definitions are then in conflict with each other in the sense that it is unclear which is the actual behavior that should be used.

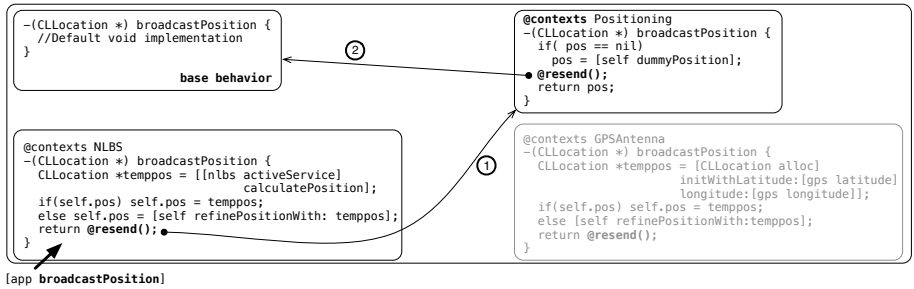
To solve this question, Subjective-C implements two *method resolution strategies*. Method resolution strategies give a concrete composition order for behavioral adaptation definitions in the presence of multiple active contexts. Such strategies have an impact in the way methods are chosen. The Subjective-C method look-up mechanism supports two method resolution strategies.

The first implemented method resolution in Subjective-C is the *activation order* technique. This technique provides a dynamic order of behavioral adaptation definitions as their associated contexts become active. This ordering is based on the idea that those contexts that have been activated recently are more relevant than those that have been activated at an earlier time. Subjective-C assigns an *activation timestamp* to a context whenever this is activated. Behavioral adaptation definitions are ordered by timestamp. When a method is called, the first observable behavior is that provided by the most recently activated context. In case of behavior reuse, the next observable behavior will be that provided by the context activated immediately before that, and so on, until the base behavior of the system is reached.

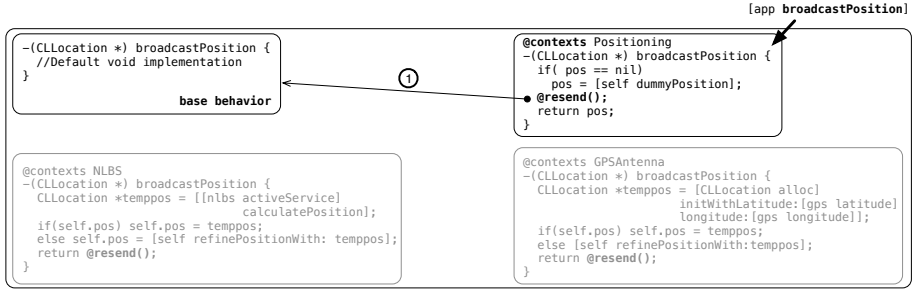
Example 4.2. (**Activation timestamp**) In the maps application, take the setting where no context is active. Whenever the `broadcastPosition` method is called, the system executes the base method definition of the method. Figure 4.4a shows this. In that setting we activate the `NLBS` context. As a consequence of such activation the `Positioning` context is activated, due to the strong inclusion dependency between the contexts. Each of these contexts provides a behavioral adaptation definition for the `broadcastPosition` method (Snippet 4.7). As the contexts are activated, a timestamp is given to each of them, first to `Positioning` and then to `NLBS`, as described in the context composition process (see Example 4.1). Since context `NLBS` is activated more recently its associated behavioral adaptation for the `broadcastPosition` method is the first observable behavior. Behavior is reused in the `NLBS` context by sending a `@resend()` message. Sending such a message generates a call to the behavioral adaptation associated with the `Positioning` context. Figure 4.4b shows this process. Numbers next to the arrows denote the order in which behavioral adaptations are called until the base behavior is reached.



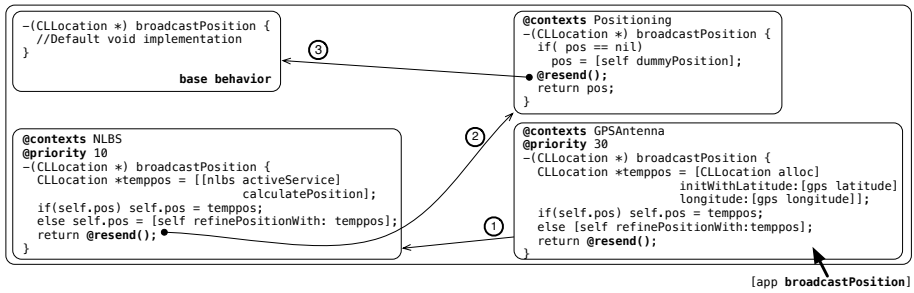
(a) Method call with no active contexts



(b) Method call with multiple active contexts and timestamps



(c) Method call with one active context



(d) Method call with multiple active contexts and method priorities

Figure 4.4: Subjective-C behavioral adaptations composition ordering.

There are situations in which a predefined (even if dynamic) method ordering based on activation timestamps of contexts is insufficient to achieve the desired behavior of the software system. For example, in the maps application given multiple positioning provider services such as `GSMLocation`, `GPSAntenna`, or `NLBS`, we would like to *always* order them by reliability of the positioning service, taking the most reliable first. To provide this behavior, Subjective-C implements a second method resolution technique, *method priorities*. This technique consists of annotating methods with a priority value dictating the order in which they should be executed. To give priority to a method, Subjective-C uses the `@priority` *value* keyword annotation. In the presence of method priorities, behavioral adaptations are always composed based on the method priorities. The method priority composition orders methods by their priority decreasingly, partial methods for which no priority is specified are assumed to have the lowest priority. If two behavioral adaptations have the same priority they are ordered according to their activation timestamps.

Example 4.3. (**Explicit priorities**) In the maps application, suppose there is only one active context in the application, context `Positioning`. The behavioral adaptations associated with the `Positioning` context are visible to the system, because behavioral adaptations are always given priority over base method definitions. The reasoning behind this is that behavioral adaptations are supposed to be more appropriate than the base behavior of the system, according to its surrounding execution environment. For example, a call to the `broadcastPosition` method in the maps application, is always resolved by one of the available behavioral adaptations (the behavioral adaptation associated with the `Positioning` context in Figure 4.4c). Figure 4.4c illustrates this method calling process when context `Positioning` is deactivated, its behavioral adaptations are withdrawn from the system and thus no longer accessible.

Let us define priorities for the `broadcastPosition` behavioral adaptations, where the behavioral adaptation with highest priority is that associated with the `GPSAntenna` context.[5] In this situation, regardless of the order in which the contexts are activated, the first executed method is that associated with the `GPSAntenna`. The following observable behavior would be the method associated with the `NLBS` context, and so on. Figure 4.4d shows the method execution ordering for all active contexts in the system. The numbers decorating the arrows show the execution order.

4.4 Subjective-C Internals

In Section 4.3 we provided an overview of the different properties of COP languages (**COP.1**–**COP.11**) supported by Subjective-C.This section provides a detailed view about Subjective-C's internals [123, 153], explaining how it works

[5] In Subjective-C the methods with highest priority are those annotated with the highest priority value.

and illustrating some of the behavioral inconsistencies we have detected in the language.

4.4.1 Architecture and Implementation

Subjective-C's implementation follows a modular architecture proposed for contex awareness in ambient intelligence and ubiquitous systems [74, 27]. Nonetheless, Subjective-C does not deal with distribution and remote interaction with external devices. Rather, Subjective-C is only concerned with the dynamic adaptation to contextual situations. This can be seen in the general architecture for context-awareness of Figure 4.5. Subjective-C currently provides an implementation for the three highlighted modules, for which we explain their purpose and internal design in the following.

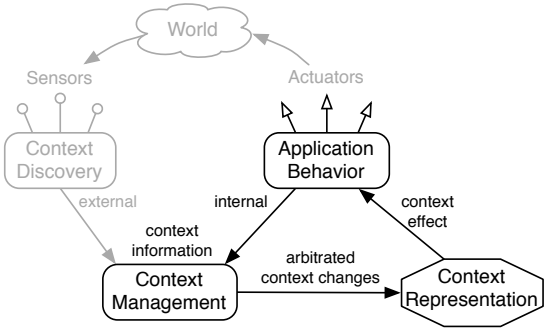


Figure 4.5: General context-awareness architecture [27] of Subjective-C.

Context Representation Module

The context representation module is used to provide a concrete representation of the ensemble of contexts defined in the system. Such an ensemble is used to oversee the relations and interactions between different context objects, provide a direct access to behavioral adaptations, and query and modify the state and behavior of contexts. It is common to embody the context representation as a dedicated data structure of the system. However, the COP languages [74, 77, 98, 163] that provide an explicit representation of contexts as first class entities of the system define two instantiations of the context representation (two data structures): an instantiation of all contexts defined in the system, which we will refer to as the *context dependency graph*, and an instantiation consisting of the subset of contexts that are currently active, which we will refer to as the *active context* of the system.

Subjective-C takes inspiration from the representation of contexts as a context graph introduced in Ambience [75]. Subjective-C represents contexts by means of context dependency graphs. A graph consisting of contexts as nodes, and dependency relations as edges. Each of the edges in the graph has a type

representing the semantics of the interaction between contexts (e.g., exclusion or requirement).

In Subjective-C the context dependency graph is expressed by means of adjacency lists.[6] Each context contains a list of all of its context dependency relations —that is, a list of triplets $< R, c_1, c_2 >$, where R is the type of the context dependency relation, and c_1 and c_2 are the two context objects interacting through the context dependency relation. The Subjective-C context graph is a directed graph, this means that edges represented by the triplets $< R, c_1, c_2 >$ and $< R, c_2, c_1 >$ are different. Every context has a reference to all its context dependency relations. This means that for every dependency relation R defined between two contexts c_1 and c_2, the triplet $< R, c_1, c_2 >$ is associated with the two contexts. For example, in the maps application, if we assume a weak inclusion to be of type C, the edge $< C, \texttt{WLAN}, \textsc{Connectivity} >$ will be an entry in the adjacency list of both the `WLAN` and CONNECTIVITY contexts.

In the implementation of Subjective-C, the set of active context is represented as a list. Whenever a context is activated, it is added to the list of active contexts (if it was not there before). Whenever a context is deactivated (if its activation counter is zero), it is removed from the active contexts list.

Application Behavior Module

The application behavior module, is responsible for providing the behavior of the system after behavioral adaptations are composed. As mentioned earlier in this section, the context representation module, and in particular the active context, can be used to decide the behavior to be executed when a method is called. Section 4.3.3 introduced the composition of behavioral adaptations in Subjective-C. Behavioral adaptations are composed according to the order given by the available disambiguation techniques. The observed behavior of the system is that provided by the first method of the ordering.

Instead of literally introducing and withdrawing methods, Subjective-C uses a special technique for the dynamic adaptation of system behavior, namely *method swizzling*,[7] or method replacement. The idea behind method swizzling is that the implementations the base method and its behavioral adaptation are interchanged by swapping their pointers, whenever contexts are activated. To accomplish this, all behavioral adaptations defined in Subjective-C are "pre-loaded" with the rest of the application. Behavioral adaptations and their corresponding base behavior are implemented as Subjective-C methods following the definition given in Snippet 4.9. That is, each behavioral adaptation is characterized (among others) by its selector (Line 4), implementation (Line 11) and a list of the related behavioral adaptations (Line 7).

The process of "pre-loading" behavioral adaptations in Subjective-C is described as follows: (1) Subjective-C precompiles the application before deploy-

[6] `http://www.ics.uci.edu/~eppstein/161/960201.html`.

[7] `http://cocoadev.com/wiki/MethodSwizzling`.

ing it onto a mobile device, (2) the pre-compilation phase is in charge of gathering all method definitions annotated by a `@contexts` keyword, (3) each behavioral adaptation is renamed by prepending the name of the context to which is associated, (4) the respective Subjective-C method is created with such a name (following the skeleton of Snippet 4.9), finally (5) these methods are then compiled with the rest of the application using the regular Objective-C compiler.[8]

```
1  @interface SCMethod : NSObject {
2  @public
3    Method methodStructure;
4    SEL selector;
5    Class relatedClass;
6    BOOL instanceMethod;
7    NSMutableArray *contextMethods;
8    SCDispatchMode dispatchMode;
9    SCArgProcessor *argsProcessors;
10   NSUInteger numberOfArguments;
11   IMP localDispatcher;
12   NSMethodSignature *methodSignature;
13   NSMutableSet *contextsSet;
14 }
```

Snippet 4.9: Subjective-C method definition.

Every time a context is activated in the system, its associated behavioral adaptations replace the implementations of the base methods by swizzling the behavioral adaptation implementation with that of the base method, or the implementation corresponding to the active context. Then the new implementation registers the old implementation as the *next method* to be called according to the given order. Normally this is the first behavioral adaptation in the order before the context activation (unless method priorities are used). Whenever `@resend()` is called within a method, a swizzling process takes place replacing the current method implementation with that registered in the `SCMethod` as the next method, and the method is called again. After this call is made the methods are swizzled back into place.

Similarly, every time a context is deactivated in the system, its associated behavioral adaptations are withdrawn from the system by swizzling their implementation with that of the next method, so that the behavioral adaptation is not linked anymore to the method ordering of the base behavior.

Context Management Module

The purpose of the context management module is to orchestrate context activations, so that changes in the surrounding execution environment do not lead to behavioral inconsistencies. Context orchestration is realized by *adaptation policies* [50]. Adaptation policies lay down the conditions for which context activations can or cannot be performed taking into account the *current context* of the application. Whenever a context activation takes place, adaptation policies are first checked for satisfiability (i.e., the system makes sure that the context

[8] The LLVM 2.0 compiler for the release 2.0 of Subjective-C.

activation does not violate any of the adaptation policies, leading to contradictory behavior). Second, adaptation policies may associate context activations with specific actions to be performed.

Previous COP language implementations propose approaches similar to the use of a context management system [50, 51, 76]. However, Subjective-C is the first language where the use of a context manager has been made explicit. Currently the context management module of Subjective-C consists of two tightly related submodules. The first module is in charge of structuring contexts based on the definition of context dependency relations as a reification of adaptation policies. The second module is in charge of updating context states as described by the context dependency relations. Currently Subjective-C manages context interaction through the four context dependency relations described in Section 4.3.3, where every context is aware of the contexts related to it. For example, in the maps application the context `NLBS` contains a list of two elements (the requirement dependency relation `NLBS`–◀`Connectivity` and the strong inclusion dependency relation `NLBS`–▶`Positioning`).

When an `@activate()` or `@deactivate()` message is sent to a context in order to respectively activate or deactivate it, the context is not immediately activated, rather the context activation is requested. The context activation is effectively performed if and only if all of the constraints imposed by all of the context dependency relations of the context are validated. Every context activation is received by the context manager and forwarded to be resolved by the specific context object. The context object is activated only if none of its context dependency relations restrict its activation. For example, a strong inclusion dependency relation activation of the source context requests the activation of the target context. In such a case, the context activation can take place only if all of the forwarded activation requests are also valid. The process of activating a context part of a strong inclusion dependency relation is shown in Snippet 4.10.

Example 4.4. Snippet 4.10 illustrates the process for context activation. In the maps application, assume that context `Connectivity` is active due to a `WLAN` network being available. Suppose further that we activate the `NLBS` context. When the `@activate(NLBS)` message is sent for the activation of `NLBS`, this message is resolved by the context manager through the method defined in Line 2, which forwards the request to activate the context to the object representing that context, by calling the method in Line 6. This method is in charge of activating the context if and only if it can be activated, a verification which is carried out by the method described in Line 18. Note that Lines 25 to 26 check the possibility to activate the context for all of its context dependency relations. In the case of `NLBS` this means checking its requirement and strong inclusion dependency relations. The method in Line 30 shows the process of verifying the activation for a context dependency relation of type strong inclusion.[9] Processing the strong inclusion dependency relation of

[9] The process to verify the deactivation, as well as the verification of the other context dependency relation types is similar.

```
1  //Receiver method of @activate() in the context manager
2   - (BOOL)activateContext:(SCContext *)aContext {
3     return [aContext activate];
4   }
5  //Activation methods of context objects
6   - (BOOL)activate {
7     //One activation at a time
8     @synchronized([SCContext class]) {
9       if (![self canBeActivated])
10          return NO;
11      NSMutableSet *contextsSet = [[NSMutableSet alloc] init];
12      BOOL canActivate=[self activateExceptContexts:contextsSet];
13      [contextsSet release];
14      return canActivate;
15    }
16  }
17 //Verification of the conditions for the exclusion dependency relation
18  - (BOOL)activateExceptContexts:(NSMutableSet *)contextsSet {
19    NSParameterAssert(contextsSet != nil);
20    if ([contextsSet containsObject:self])
21      return YES;
22    [contextsSet addObject:self];
23    self.activationCount + = 1;
24    [self processActivation];
25    for (SCDependencyRelation *aLink in links)
26      [aLink processSourceActivationExcept:contextsSet];
27    return YES;
28  }
29 //Process activation for the strong inclusion dependency
30  - (BOOL)processSourceActivationExcept:(NSMutableSet *) contextSet {
31    if (![target activateExceptContexts:contextSet])
32      return NO;
33    return YES;
34  }
```

Snippet 4.10: Context activation method in Subjective-C.

context NLBS forwards the request of activation to the target context (in this case Positioning), as shown in Line 31. Notice that now we have to verify if the activation of context Positioning is possible by means of method activateExceptContexts in Line 18. All dependency relations defined for the Positioning context (GSMLocation–▶Positioning, NLBS–▶Positioning, GPSAntenna–▶Positioning, Private□–□Positioning) must be verified in order to decide whether the activation of Positioning is possible or not. The definition of the strong inclusion dependency relation does not impose any restrictions on the activation of the target context, so it is only necessary to verify that the Private context is inactive. Since by assumption the only active context in our initial configuration for the example was Connectivity, context Positioning can be (and is) activated. Secondly the requirement dependency NLBS–◀Connectivity is to be verified. The definition of the requirement dependency relation states that the source context of the relation can be activated only if the target context is already active. Since we assumed Connectivity to be active, context NLBS can thus be activated. As the conditions imposed by all of the context dependency relations associated with NLBS are verified, the context can be effectively activated.

Note that in the case context Connectivity is inactive, the activation of NLBS

would be denied, denying the activation of all the contexts requested to be activated in the process described here, in particular not activating POSITIONING.

4.4.2 Programming Support

Throughout Section 4.3 we showed how Subjective-C modularizes, selects, scopes, and composes adaptations by means of the definition of context objects, context activation and context interaction. In this section we present a synopsis of the programming facilities provided by Subjective-C to aid the development of COP systems. Table 4.3 summarizes the minimal set of language facilities used in Subjective-C in order to enable dynamic adaptation to context. Comprehensive examples on how to use these (and similar) constructs in practice can be found in Chapter 9.

Context declaration ::= **@context(** *context-name* **)**
Context activation ::= **@activate(** *context-name* [**in** *thread-name* {**,** *thread-name* }] **)**
Context deactivation ::= **@deactivate(** *context-name* [**in** *thread-name* {**,** *thread-name* }] **)**
Context method annotation ::= **@contexts** *context-name* { *context-name* }
Method priority declaration ::= **@priority** *priority*
Behavior reuse ::= **@resend()**
Dependency relations declaration ::=
[addExclusionBetween: *context-name* **and:** *context-name***]** |
[addWeakInclusionFrom: *context-name* **to:** *context-name***]** |
[addStrongInclusionFrom: *context-name* **to:** *context-name***]** |
[addRequirementTo: *context-name* **of:** *context-name***]**

Table 4.3: Subjective-C syntax for Context-Oriented Programming.

In addition to these COP language abstractions, Subjective-C provides a Domain-Specific Language (DSL) to facilitate the definition of contexts and context dependency relations. The Extended Backus-Naur Form (EBNF) of the context declaration DSL is shown in Table 4.4. The definition of context objects and their context dependency relations is given by providing a list of context names, and a list of context dependency relations between those contexts. Context dependency relations can be defined between two or more contexts. For example, $E \Rightarrow F$ defines a strong inclusion dependency E–▶F, whereas $(A\ B) \Rightarrow (C\ D)$ defines the strong inclusion dependencies A–▶C, A–▶D, B–▶C, and B–▶D.

Context Declaration File	::=	**Contexts:** { *ContextName* } **Links:** { *DependencyDefinitions* } **END**
ContextName	::=	*context-name*
DependencyDefinitions	::=	**(** {*ContextNames*} **)** *DependencyConnector* **(** {*ContextNames*} **)** | **><(** *ContextNames* **)**
DependencyConnector	::=	**->** | **=>** | **=<** | **><**

Table 4.4: Subjective-C DSL syntax.

To illustrate this syntax, Snippet 4.11 shows the contexts and context dependency relations defined in the maps application, where Line 11 defines an exclusion, Lines 12 and 13 define a weak inclusion, Lines 14 through 16 definesa strong inclusion, and Line 17 defines a requirement.

```
1  Contexts:
2    Private
3    Positioning
4    NLBS
5    GPSAntenna
6    GSMLocation
7    Connectivity
8    WLAN
9    Bluetooth
10 Links:
11   Positioning >< Private
12   WLAN -> Connectivity
13   Bluetooth -> Connectivity
14   GSMLocation => Positioning
15   GPSAntenna => Positioning
16   NLBS => Positioning
17   NLBS =< Connectivity
18 END
```

Snippet 4.11: Contexts and context dependency relations DSL specification.

4.4.3 Behavioral Inconsistencies

In Section 4.2 we presented different situations in which inconsistencies may arise in COP systems due to the informal definition of how contexts interact when they are activated, and the mismatch between their high level representation as a context graph and their implementation. Subjective-C was developed with the intention of avoiding inconsistencies in the behavior of COP systems. In particular, to deal with dynamicity of adaptations, Subjective-C implements the context representation module by means of a context dependency graph and a set of active contexts. To deal with the interaction of adaptations, Subjective-C implements a context management module by introducing context dependency relations. To deal with multiplicity of adaptations, Subjective-C uses activation counters. However useful all these are, the language still leaves room for some behavior inconsistencies. In this section we discuss the inconsistencies specific to the programming model of Subjective-C.

Subjective-C provides a first attempt to manage behavior consistency. Subjecti C, in particular, has three behavioral inconsistencies that were overlooked by the designers and implementors of the language.

The first inconsistency is present in the deactivation of contexts in a strong inclusion relation, as follows:

Example 4.5. (**Strong inclusion deactivation cycle**) Consider the interaction between contexts in a strong inclusion dependency relation in the maps application (i.e., GPSAntenna–▶Positioning). This means that activation of the former context implies activation of the latter (as soon as there is GPS signal reception, the positioning service becomes automatically available), and

deactivation of the latter implies deactivation of the former (when the positioning service is turned off, there is no longer a need for GPS signal reception which is turned off, for example, to save battery). Similarly, GSMLocation and NLBS also strongly include Positioning. Table 4.5 shows the original specification of these interactions between the source and target contexts.

message	source behavior	target behavior
canActivate	target canActivate	YES
canDeactivate	YES	source canDeactivate
activate	target activate	—
deactivate	target deactivate	source deactivate

Table 4.5: Strong inclusion relation specification [77].

Suppose that both GSMLocation and GPSAntenna are activated. This will cause two activations of Positioning, because of the activation counters. Following the semantics of strong inclusion, deactivation of any concrete location service, for instance GSMLocation, should cause *one* corresponding deactivation of Positioning. Context Positioning should still have a remaining activation coming from GPSAntenna. However this is not the case as it can be seen by the last row of Table 4.5. The implementors of Subjective-C initially overlooked multiple activation in their informal definition of the strong inclusion dependency relation, which caused context Positioning to be deactivated immediately in the previous scenario, even though in theory it should have remained active.

A second inconsistency, also present in the strong inclusion dependency relation, exists whenever different contexts strongly include the same context, as follows:

Example 4.6. (**Strong inclusion accidental interaction**) Consider again the case of the Positioning context of the maps application. Such a service can be made available, for example, via the activation of the GPSAntenna of the device (GPSAntenna–►Positioning), or the use of the mobile network via the GSMLocation context (GSMLocation–►Positioning). Let us assume now the situation where both the GPSAntenna and GSMLocation contexts are active, for example by executing @activate(GPSAntenna) and @activate(GSMLocation). According to the semantics of the strong inclusion dependency (Table 4.5), each time the source context is activated, the activation request is forwarded to the target. So the state of the system after the two activations will consist of GSPAntenna and GSMLocation activated one time each, and Positioning activated 2 times. Further, let us suppose that we activate the positioning context independently @activate(Positioning), so that context is activated 3 times. Now suppose that the connection with the GPSAntenna is lost, for example because the user enters a building, so that a request to deactivate the context is sent. We know from Example 4.5 that both Positioning and

GPSAntenna will be inactive. However, since the deactivation of Positioning requests the deactivation of all of its sources, additional requests to deactivate context GSMLocation will be sent, causing it to be deactivated unintendedly. In theory the GSMLocation context should remain active (just as the Positioning service should remain active) since there is no relation between it and GPSAntenna. The deactivation of GPSAntenna should be independent of that of the GSMLocation context, as it is the case for their activation.

The final design error in Subjective-C due to the informal specification of context dependency relations can be seen in the inconsistent behavior of the requirement dependency relation.

Example 4.7. (**Required context deactivation**) Consider the interaction between contexts in a requirement dependency relation in the maps application (i.e., NLBS–◀Connectivity). This means that the former context can only be activated if the latter has been previously activated (a near location base service can only be provided if there is a short range network connection available). Consequently if the latter context is no longer active, the former should be deactivated (when connectivity is turned off, the available services needed for NLBS are no longer available and the service should become unavailable).

message	source behavior	target behavior
canActivate	target isActivate	YES
canDeactivate	YES	source canDeactivate
activate	—	—
deactivate	—	source deactivate

Table 4.6: Requirement relation specification [77].

Suppose that contexts WLAN and Bluetooth are active. Since these two context have a weak inclusion dependency with Connectivity, this context is activated twice. Suppose further that the NLBS context is activated (e.g., by directly turning the service on). If the user loses connectivity of the device because the Bluetooth connection is no longer in range, a deactivation is requested to the Connectivity context. This deactivation also triggers the deactivation of the NLBS context. However, following the semantics of the requirement dependency relation, the NLBS context is only deactivated if Connectivity is inactive, which is not the case since WLAN is still active. The implementors of Subjective-C initially overlooked multiple activations in their informal definition of the requirement dependency relation, which caused triggering the deactivation of context NLBS when Connectivity was deactivated (shown in the last row of Table 4.6), even though in theory it should remain active.

Examples 4.5 through 4.7 provide a compelling illustration of the need for a formal specification of COP systems and in particular of the interaction between

contexts. Such formal specification could be used to define a notion of system consistency and to ensure that a particular system is indeed consistent. Chapter 6 presents a model for such a specification, which beyond serving as formal foundation is also implemented and integrated with the host programming language to serve as run-time mechanism for the management of consistency in Dynamically Adaptive Software Systems.

4.5 Conclusion

In this chapter we presented context-aware systems as a particular class of Dynamically Adaptive Software Systems, which are characterized by their high dynamicity and the fine granularity of the software adaptations they provide. The chapter presents the Context-Oriented Programming (COP) paradigm, a programming paradigm tailored to the development of context-aware systems.

COP systems were selected as the focus of our study because they satisfy all the requirements (alongside self-adaptive systems) defined for Dynamically Adaptive Software Systems in Chapter 2. Timelines (**D.1**) is satisfied by the selection characteristic of COP. Context activation takes place unannounced as a consequence of changes in the surrounding execution environment of the system. Granularity (**D.2**) is satisfied by the possibility of adapting entities of the system at different levels of granularity. Here we only discussed the adaptation of methods; however, languages like ContextL or Ambience also allow the adaptation of fields or even complete objects. Flexibility (**D.3**) is satisfied by the modularization characteristic of COP systems. Adaptations are defined independently from the base system. Compatibility (**D.4**) is satisfied by means of the activation constructs provided by COP languages. Every time there is a change in the surrounding execution environment of the system, the context reifying the situation change is activated. Independence (**D.5**) is satisfied by the modularity characteristic of COP systems. Each adaptation is defined as an independent module of the system which can be deployed at any time during the execution of the system.

To provide a more detailed description of COP systems, we focussed on four characteristic properties which describe the dynamicity and modularity of software adaptations in COP systems: **(1)** modularity of adaptations (Section 4.1.1), which describes how adaptations are defined and structured (at the programming level) with respect to the base logic of the system, **(2)** selection of adaptations (Section 4.1.2), which describes how adaptations are selected to be included or withdrawn from the system, **(3)** scoping of adaptations (Section 4.1.3), which describes the period during which an adaptation is available, and **(4)** composition of adaptations (Section 4.1.4), which describes how adaptations interact with each other and how behavioral adaptations are composed between each other and the base logic of the system. The properties presented for each of the four characteristics were extracted from currently existing COP languages, and were rendered explicit in an incarnation of one of such languages, Subjective-C, which is used as the language laboratory throughout this

dissertation.

In addition to these four characteristics of COP systems, we presented three characteristics of COP systems (dynamicity, interaction, and multiplicity of adaptations) that can give rise to behavioral inconsistencies or unpredictability due to the dynamicity of adaptations. With these in mind we presented current approaches to manage consistency in COP languages, providing an overview of their strengths and shortcomings.

Besides the analysis of consistency management in existing COP languages (Table 4.2), we pinpointed two problems common to all approaches: the lack of a formal definition of adaptation interactions, and the obligation for developers to manually verify if the constraints established for adaptation interaction are safe.

In the following chapters we introduce a formal model that can be used for the specification and management of dynamic adaptations, which will evolve in Chapter 6 into the development of a theory setting the foundations for the definition of consistent Dynamically Adaptive Software Systems.

Chapter 5

Petri Nets

In Chapter 3 we discussed the main problems for the realization of Dynamically Adaptive Software Systems. In particular we discussed different models that could be used for maintaining the consistency of software systems in general, and the shortcomings of such models in a dynamic setting. In this chapter we present a descriptive model of software systems that provides a sound representation of the system's structure and dynamics. This model complies with the conflict resolution model requirements (**M.**1 through **M**.4) we presented in Chapter 2 for the consistency management of Dynamically Adaptive Software Systems.

The proposed model is the Petri net model. Petri nets [144] have been extensively used to model different kinds of systems ranging from chemistry processes to communication processes or synchronization control of software systems. Petri nets provide a mathematical representation of the systems they model. Using this representation, systems can be analyzed to reveal information about their structure and dynamics. The advantage of Petri nets over other modeling approaches is that they cover the specification, execution and analysis of the system within the same formalism. Petri nets were introduced as an answer to other modeling approaches, such as automata or transition systems, that could not describe the data flow of systems. Petri nets deal with the interaction of concurrent components in computing systems, where interactions between components can occur non-deterministically. In the setting of Dynamically Adaptive Software Systems, dynamic adaptations to the system behavior or interaction between system components can take place at any moment in time, thus these can be modeled as Petri nets. For this reason, Petri nets seem to be a good fit to model Dynamically Adaptive Software Systems, and in particular COP systems, where the system's behavior adapts to events or situations in its surrounding execution environment.

In this chapter we provide a general introduction to Petri net theory, we also explain some extensions to the basic model which have been used for the modeling of specific concerns in different domains of computing. This chapter

is rounded off by a discussion on how these extensions can be used for the modeling of dynamically adaptive systems, and in particular, context-aware systems.

5.1 Introduction to Petri Nets

Petri nets are defined as directed bipartite graphs, with *places* and *transitions* as disjoint node sets [156, 137].

Definition 5.1. *A **Petri net** is a quadruple $\mathcal{P} = \langle P, T, f, m_0 \rangle$, where: P is a finite set of places, T is a finite set of transitions, and $P \cap T = \phi$, $f : (P \times T) \cup (T \times P) \longrightarrow \mathbb{Z}^*$ is the flow function, and m_0 is a function assigning tokens to places which describes the initial marking of the Petri net $m_0 : P \longrightarrow \mathbb{Z}^*$.*[1]

The flow function defines the number of tokens that pass through between a place and a transition, and *vice versa*. There cannot be any arcs between two places or two transitions. A marking is a distribution of *tokens* over the places representing the state of the system. Intuitively, the tokens described by the initial marking start to flow through the network according to the arcs described by the flow function, yielding a new marking at every step. The marking function allows for multiple tokens to be assigned to a single place.

Figure 5.1 shows an example of a Petri net $\mathcal{P}$, where $P = \{p_1, p_2\}$, $T = \{t_1, t_2, t_3\}$, $m_0(p_1) = 2$, $m_0(p_2) = 0$, and the flow function f is defined by the table in the left-hand side of Figure 5.1. The rows in this table correspond with the first argument of the flow function, whereas the columns correspond with the second argument of the flow function. For example, for row t_2 and column p_2, $f(t_2, p_2) = 2$, meaning that there are 2 arcs from transition t_2 to place p_2. A zero-entry in the table, for example, $f(p_1, t_1) = 0$ means that there are no arcs between place p_1 and transition t_1. Arcs in the right-hand side of Figure 5.1 are labeled with their weight —that is, the number of tokens that they carry. To increase the readability of Petri net visualizations, if arcs have a weight of 1, the corresponding label will be omitted from figures hereafter.

f	t_1	t_2	t_3	p_1	p_2
p_1	0	1	0	-	-
p_2	0	0	1	-	-
t_1	-	-	-	1	0
t_2	-	-	-	1	2
t_3	-	-	-	0	0

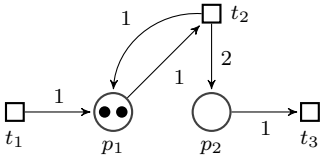


Figure 5.1: Petri net. Definition of its flow function (left), and visual representation (right).

[1] $\mathbb{Z}^*$ represents the set of non-negative integers

Places usually represent conditions or states of a system, like "*battery is low*". A state is valid (active) if there is at least one token in the respective place. *Transitions* usually represent events or actions to be performed on states. Transitions modify the state of a system by the flow of tokens from one place to another after firing the transition. Given a transition t, its set of *input places* is defined by $\bullet t = \{p_{in} \in P \mid f(p_{in}, t) > 0\}$, similarly, its set of *output places* is defined by $t\bullet = \{p_{out} \in P \mid f(t, p_{out}) > 0\}$.

Definition 5.2. *A transition $t \in T$ is enabled at a marking m, written $m[t\rangle$, if and only if $\forall p_{in} \in P$ such that $p_{in} \in \bullet t,\ m(p_{in}) \geq f(p_{in}, t)$.*

Once transitions become enabled, they may fire. Firing transitions (also known in the literature as *transition triggering* [61] or the *token game* [144]) describes the dynamics of the Petri net by modifying the marking.

Definition 5.3. *A transition firing modifies marking m of a Petri net to a new marking m', written $m[t\rangle m'$. Firing of transition modifies the state of the Petri net such that $\forall p \in P,\ m'(p) = m(p) - f(p,t) + f(t,p)$.*

In the example of Figure 5.1, firing transition t_2 will yield a new marking m_1, from m_0, where $m_1(p_1) = 2$, $m_1(p_2) = 2$, that is, $m_0[t_2\rangle m_1$.

Definition 5.4. *A* step*, or firing step, Υ is defined as a sequence of transition firings. If the sequence of transitions is finite, that is, $\exists\, t_0, \ldots, t_n \in T$, and $m_1, \ldots, m_n$ markings of $\mathcal{P}$, such that $m[t_0\rangle m_1[t_1\rangle \ldots m_n[t_n\rangle m'$ for two markings m and m', we say that m' is **reachable** from m via step Υ, and write $m[\Upsilon\rangle m'$.*

Note that multiple transitions may be enabled in a Petri net at any given moment in time. Being a non-deterministic model, Petri nets allow firing any of them.

Transitions such as t_1 in Figure 5.1 with no input places are called *sources*. They are always enabled. Transitions such as t_3 in Figure 5.1 with no output places are called *sinks*. Tokens are removed from the Petri net after their firing.

Resources in a system are generally finite, to represent a scarce resource, like "*buffer capacity is 512KB*." Petri nets introduce the concept of *place capacities*. Petri nets with place capacities allow us to define the maximum number of tokens (resources) that a place can hold.

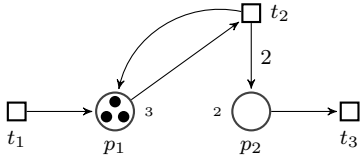


Figure 5.2: A Petri net with place capacities.

Definition 5.5. *A Petri net with place capacities is defined as a 5-tuple* $\mathcal{P} = \langle P, T, f, \mathcal{K}, m_0 \rangle$*, where* $\mathcal{P} = \langle P, T, f, m_0 \rangle$ *is a regular Petri net, and* $\mathcal{K} : P \to \mathbb{Z}^*$ *is a function assigning a capacity* $k \in \mathbb{Z}^*$ *to every place* $p \in P$*. The capacity function represents the restriction that a place* $p \in P$ *can hold at most* k *tokens. A Petri net* $\mathcal{P}$ *is said to have a* k*-capacity if there exists a* $k \in \mathbb{Z}^*$ *such that* $\forall\, p \in P,\ \mathcal{K}(p) \leq k$*.*

Capacities are assigned statically to Petri nets —that is, once a place is given a capacity, this will be set forever. The use of place capacities have as a consequence a modification in the semantics of firing rules for Petri nets. A transition t is said to be enabled for a marking m, $m[t\rangle$, if and only if:

1. $\forall\, p_{in} \in {\bullet t},\ m(p_{in}) \geq f(p_{in}, t)$.
2. $\forall\, p_{out} \in t{\bullet},\ m(p_{out}) + f(t, p_{out}) - f(p_{out}, t) \leq k$.

Figure 5.2 illustrates an example of a 3-capacity Petri net. In this figure place capacities are the small numbers decorating the places. p_1 has a capacity of 3 and p_2 has a capacity of 2. If a place does not have an associated capacity, then it can hold as many tokens as desired. Note that transition t_1 in Figure 5.2 is not enabled since its output place p_1 has reached its maximum capacity. In the remainder of this text we will treat places as having infinite capacity unless explicitly stated otherwise.

5.1.1 Petri Net Properties

Providing a formal model of a system just for the sake of modeling or formalizing the system is of little utility. The purpose of having a conceptual model of a system is to be able to abstract the system and reason about its properties. Petri nets find their strength in providing different analysis techniques about the system's structure and its behavior [143]. We discuss here the most prominent behavioral properties of Petri nets that could be used in the context of Dynamically Adaptive Software Systems. An unabridged list of the structural and behavioral properties can be found in [137].

Reachability and Coverability

The reachability and coverability properties study the different states in which the system could be, based on its initial state (initial marking). Reachability can be used to study if specified resources of the system will be in use simultaneously or not. The reachability problem in Petri nets is concerned with deciding if given a marking of the system, that marking is reachable from the initial marking.

Coverability is used to study whether the availability of a particular resource in the system is sufficient or not for a given Petri net state or not. The coverability problem is closely related to the reachability problem in the sense that it is also about finding reachable markings of the system. The coverability problem is concerned with given a reachable marking m from the initial marking

of the Petri net, finding other reachable markings m' from the initial marking covering m such that the number of tokens at state m' is greater or equal than the number of tokens at state m for every place of the Petri net. If there is a marking covering the desired capacity of a given resource, we will say that the resource is sufficiently available.

Definition 5.6. *Given a marking m' of a Petri net $\mathcal{P}$, m' is said to be **coverable** if there exist a marking m reachable from the initial marking, such that $\forall\, p \in P,\ m(p) \leq m'(p)$.*

Liveness

The liveness property is used to study the presence of deadlocks in a system, that is, to analyze if certain resource can never be available (resp. freed) because its associated input (resp. output) transitions can no longer fire from a given reachable marking on.

Definition 5.7. *Given a Petri net $\mathcal{P}$, we say that $\mathcal{P}$ is **live** if and only if $\forall\, m$ reachable marking from the initial marking and $\forall\, t \in T$, there is a finite step Υ, where $m[\Upsilon\rangle m'$ such that $m'[t\rangle$.*

Coupled with the notion of liveness is the notion of deadlocks. A **deadlock** occurs in a system if for a reachable marking m, there are transitions in the Petri net which never become enabled. This is defined formally in Definition 5.8.

Definition 5.8. *Given a Petri net $\mathcal{P}$, we say that a transition $t \in T$ is **dead** if there exists no step Υ, where $m_0[\Upsilon\rangle m$ such that $m[t\rangle$.*

Liveness is an ideal property for the different actions that can be executed in the system. In our approach (cf. Chapter 7) we will use the notion of deadlocks to analyze the system actions. That is, for each of the possible actions defined in the system, we will analyze whether the action can be in a deadlock according to Definition 5.8.

Fairness

The fairness property is used to analyze the starvation of resources of the system. Different notions of fairness have been proposed in the literature [15, 173, 113, 119, 196]. Here we present two basic concepts, bounded-fairness and unconditional (global) fairness [137].

Definition 5.9. *Given a Petri net $\mathcal{P}$, two transitions $t, t' \in T$, are said to be **bounded-fair** (or B-*fair*) if and only if for any given step $\Upsilon = m[t_0\rangle m_1[t_1\rangle m_2 \ldots$ $\exists\, k \in \mathbb{Z}^*$, a bound such that t fires maximum k times before the first firing of t'. A Petri net is called B-fair if for every pair of transitions, the transitions are B-fair.*

Notation 5.1.

- *Given a step Υ, the symbol $\Upsilon_{\restriction_t}$ represents the restriction of the step to transition t —that is, a sequence of firings of transition t.*

Definition 5.10. *In a Petri net $\mathcal{P}$ a step Υ is said to be unconditionally (globally) fair if and only if $|\Upsilon| < \omega$ (the step is finite), or $\forall\, t \in T,\ t \in \Upsilon$ and $|\Upsilon_{\restriction_t}| = \omega$ (every t occurs an infinite number of times in Υ). The Petri net is said to be fair, if for every step Υ, Υ is fair.*

Persistence

The persistency property is used to analyze the control flow of the system, specifically to see if the execution of a process in the system will inhibit that of another process.

Definition 5.11. *A Petri net $\mathcal{P}$ is said to be* persistent *if and only if $\forall\, t, t' \in T$ if $m[t\rangle$ and $m[t'\rangle$ for m a marking of $\mathcal{P}$, the firing of any of the transitions does not disable the other one. In a persistent Petri net, once a transition is enabled, it remains enabled until it fires.*

Other properties of the system such as *home markings* or *safeness* can be analyzed with Petri nets. We do not present these properties in this text because they will not be used in the immediate development of our approach, although they could be integrated for future extensions.

5.1.2 Petri Net Analysis Techniques

After observing the different properties that a Petri net may present, the question at hand is, *how can we analyze a Petri net to reason about a particular property?* Different techniques exist for the analysis of Petri net properties. Some of these techniques include the reachability tree (or coverability graph), and matrix equations [143, 137, 54]. Using such techniques exploring the Petri nets for a particular behavior can become difficult as systems grow. Different reduction rules are introduced to facilitate the exploration of the Petri net properties. Here we only mention the existing techniques without going into their detailed exploration, since this falls outside the scope of this dissertation. The existing reduction rules include: *place/transition fusion* [137], *stubborn sets* [167], *symmetries* [170], *sweep-line* [115], *cycle coverage* [171], *coverability graph* [54], *attracted execution*, and *invariant-based compression* [171].

These techniques are used by specialized analysis tools in order to reason about properties of the system. Section 5.1.3 describes which properties are supported by the analysis techniques for the specific Petri net analyzer LoLA [169].

5.1.3 A Petri Nets Analysis Tool

Several tools for the analysis of Petri net properties have been proposed [177]. These vary from specific algorithms used for the validation of a single property, to general purpose tools allowing the analysis of multiple properties. This

section presents an overview of the Low Level Petri net Analyzer (LoLA) [169],[2] an analysis tool widely used in the Petri net community for the comprehensive analysis techniques it provides, as well as for its extensibility and integration with other tools. In Chapter 7 we use LoLA to reason about properties of Dynamically Adaptive Software Systems in order to identify any inconsistencies that they may contain.

	Liveness	Reachability	Coverability	Safeness	Boundedness	Deadlocks	Reversibility	Home Marking
Place/Transition Fusion [137]	■			■	■			
Stubborn Sets [167]	■	■	■	■	■	■	■	■
Symmetries [170]	▒	▒	▒		▒	▒		▒
Sweep-line Method [115]		■				■		
Cycle Coverage [171]		■				■		
Coverability Graph [107]					▒	■		
Attracted Execution			■			■		
Invariant Based Compression [171]	■	■	■	■	■	■	■	■

Table 5.1: Supported reduction rules for each available analysis in LoLA.

We chose LoLA as a supporting analysis tool because it supports the analysis of different properties of a Petri net, hence extending the possibility to reason about different aspects of Dynamically Adaptive Software Systems. Table 5.1 shows a summary of the different Petri net properties that can be analyzed using LoLA (reachability, liveness, deadlocks, safeness [137], boundedness [62], reversibility [47], and home markings [131]) alongside the implemented reduction rules that can be (jointly) applied for such analyses. The cells marked in black in Table 5.1 represent the reduction rules applicable to the Petri net analyses. Gray cells represent properties that can be analyzed for *one single element* of the Petri net (i.e., one transition or one place).[3] For example, the symmetries reduction rule can be used when analyzing liveness for a single transition (not the complete Petri net). Analyses marked by a white cell cannot apply the reduction method.

LoLA introduces a simple (text-based) net syntax for the representation of Petri nets and the analysis to be performed. Snippet 5.1 shows the textual representation for the Petri net of Figure 5.2. Snippet 5.2 shows the definition of a particular property to be checked in this Petri net.

```
1  PLACE
2    SAFE 3: p1;
```

[2] Available at: `http://service-technology.org/lola`

[3] `http://www.informatik.uni-rostock.de/tpp/lola/documentation.htm`

```
3     SAFE 2: p2;
4   MARKING p1: 3;
5   TRANSITION SAFE t1
6     CONSUME;
7     PRODUCE p1: 1;
8   TRANSITION SAFE t2
9     CONSUME p1: 1;
10    PRODUCE p1: 1, p2: 2;
11  TRANSITION SAFE t3
12    CONSUME p2: 1;
13    PRODUCE;
```

Snippet 5.1: LoLA textual representation of a Petri net.

```
1  //Reachability - is there a reachable state where p1 is not marked?
2  ANALYSE MARKING p1: 0
3  //Liveness - will transition t3 ever be enabled?
4  ANALYSE TRANSITION t3
```

Snippet 5.2: LoLA representation of Petri net reachability and liveness analyses.

The two different specifications (a specification for reachability and a specification for liveness) shown in Snippet 5.2 are the simplest types of specification available in LoLA. Analyses of the complete Petri net are performed simply using the net definition and the desired configuration for the analysis. More complex specifications can be provided as extensively explained in by Schmidt [169].

Based on reported testing scenarios in LoLA [63, 88], the tool is able to analyze medium-sized Petri nets —that is, Petri nets of an average size of 500 places and 1000 transitions. For Petri nets of such size, the reported times of performing the analyses vary between 14 to 96 minutes depending on the type of analysis performed and the reduction rules used for each analysis.

5.2 Petri Net Model Extensions

Since their introduction in the early 60's, Petri nets have been extended with the purpose of modeling more and more complex systems. Examples among such extensions are task planning [29] or manufacturing systems [154], and providing more expressive Petri net models such as stochastic Petri nets [79], fuzzy Petri nets [141], or nested Petri nets [128], to mention some examples. Additionally, Petri nets have also been extended to serve as a modeling language for different programming paradigms, for example, OOP [18], FOP [138] and workflow management [186]. Naturally, Petri net extensions do not stop with an extension of the modeling formalism to provide a structural pattern definition and a semantics for a particular kind of system. The ultimate goal of extending the basic Petri net model is to provide specific analyses of system properties within specific domains. In this section we introduce some extensions to the structure and semantics of the basic Petri net model. These extensions become useful in Section 6.1 when we provide our own extension of the Petri net to model Dynamically Adaptive Software Systems, and in particular COP systems.

5.2.1 Static Priorities

Static priorities extend the basic Petri net model by fixing a firing order for groups of transitions [12, 16]. Establishing priorities among groups of transitions can be used, for example, for modeling systems of scheduling processes or performance analysis systems. The use of priorities explicitly sets an order for transition firing. A Petri net with static priorities is non-deterministic in the sense that within a group of transitions with the same priority, their firing occurs none-deterministically.

Definition 5.12 (**Static priorities**). *A Petri net with static priorities is defined as a 5-tuple* $\mathcal{P} = \langle P, T, f, \rho, m_0 \rangle$*, where* $\langle P, T, f, m_0 \rangle$ *is a Petri net and the function* $\rho : T \rightarrow \mathbb{Z}^*$ *decorates transitions with a weight denoting their firing order.*

Given a Petri net with static priorities $\mathcal{P}$, an ordering of its transition is given by the relation $\geq$. If $\forall t, t' \in T$ such that for a marking m $m[t\rangle$ and $m[t'\rangle$, and $\rho(t) \geq \rho(t')$, then t *always* fires before t'. If $\forall t, t' \in T$ such that for a marking m $m[t\rangle$ and $m[t'\rangle$, and $\rho(t) = \rho(t')$, then firing of t and t' is non-deterministic.

Definition 5.13. *Given a Petri net* $\mathcal{P}$ *with static priorities, a transition* $t \in T$ *is enabled at a marking* m *if and only if* $\forall p_{in} \in \bullet t,\ m(p_{in}) \geq f(p_{in}, t) \wedge \nexists\, t' \in T$*, with* $\rho(t') > \rho(t)$ *such that* $m[t'\rangle$.

An example of a Petri net with static priorities is shown in Figure 5.3. Priorities are shown as small *italic* numbers decorating each transition.

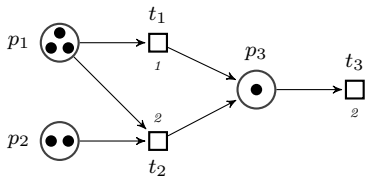


Figure 5.3: A Petri net with static priorities

In the example of Figure 5.3 both t_2, and t_3 can fire. t_1 cannot fire because it has a lower priority than both t_2 and t_3, which are enabled. Only after t_2 has fired twice and t_3 three times (regardless of the order in which they are fired), t_1 becomes enabled and can fire.

The semantics of Petri nets with static priorities are equivalent to the semantics of regular Petri nets under the conditions given by Definition 5.14.

Definition 5.14 (**EQUAL-conflict** [12]). *A Petri net with static priorities satisfies the EQUAL-conflict condition if and only if* $\forall t, t' \in T : \bullet t \cap \bullet t' \neq \phi$ *implies that* $(a)\, t, t'$ *have equal priority, and* $(b)\, \exists\, p \in \bullet t \cap \bullet t' : f(p, t) \neq f(p, t') \Rightarrow t, t'$ *both have the lowest priority.*

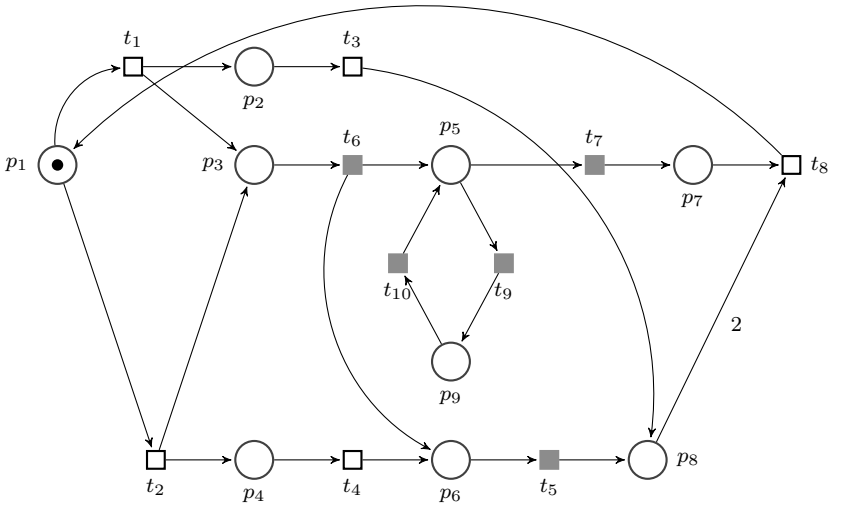


Figure 5.4: Example of Petri net with static priorities satisfying the EQUAL-conflict condition [12].

Example 5.1. To show an example of Petri net that satisfies the EQUAL-conflict condition, let us reuse the Petri net with static priorities $\mathcal{P}$, shown Figure 5.4 and originally presented in Bause [12]. The Petri net has two priority classes: $\rho_1 = \{t_1, t_2, t_4, t_8\}$ the set of transitions with lower priority represented by white squares in Figure 5.3, and $\rho_2 = \{t_5, t_6, t_7, t_8, t_9, t_{10}\}$ the set of transitions with higher priority represented by gray squares in Figure 5.4. Note that this net satisfies the EQUAL-conflict condition. The only transitions with non-empty intersection of their inputs are t_1 and t_2, and t_7 and t_9. In the first case $\bullet t_1 \cap \bullet t_2 = \{p_1\}$. Transitions t_1 and t_2 have the same priority so condition (a) is satisfied. Since $f(p_1, t_1) = f(p_1, t_2)$ condition (b) is also satisfied (by the empty condition property). In the second case $\bullet t_7 \cap \bullet t_9 = \{p_5\}$. Similarly to the previous case, transitions t_7 and t_9 have the same priority, satisfying condition (a). Since $f(p_5, t_7) = f(p_5, t_9)$, condition (b) is also satisfied.

The Equal-conflict condition is used to relate the token game semantics of regular Petri nets with that of Petri nets with static transition priorities. In Figure 5.4 let us consider the firing step $\Upsilon = m_0[t_1 t_3 t_6 t_5 t_7\rangle m$, starting from the initial marking $m_0(p_1) = 1$. Under the regular token game semantics step Υ reaches the marking m, $m_0[\Upsilon\rangle m$, where $m(p_8) = 2, m(p_7) = 1$. Nonetheless, Υ is not a firing step of $\mathcal{P}$ under the semantics of Petri nets with static priorities because, transition t_3 is fired before transition t_6, which has a higher priority. However, a permutation of Υ, $\Upsilon' = t_1 t_6 t_5 t_7 t_3$ is a firing step under the Petri nets semantics with static priorities such that $m_0[\Upsilon'\rangle m$. Theorem 5.1

generalizes the result observed in this example. The proof to this theorem can be found in [12].

Theorem 5.1. *Let $\mathcal{P}$ be a Petri net with static priorities that satisfies the EQUAL-conflict condition. Let m and m' be two markings and Υ a step such that $m[\Upsilon\rangle m'$ under the regular token game semantics. If $\forall t \in T$ such that $m'[t\rangle \Rightarrow \nexists\, t' \in T$ such that $\rho(t') < \rho(t)$, then $\exists\, \Upsilon'$ permutation of Υ such that $m[\Upsilon'\rangle$ under the semantics of Petri nets with static transition priorities.* □

5.2.2 Reactive Petri Nets

Reactive Petri nets [61] extend the basic Petri net model by introducing reactivity to it. To make the system reactive it must be opened up such that it interacts with its environment. Since Petri nets are used to model closed systems, the model is extended by allowing transitions to respond to stimuli from the system environment, and by allowing the automatic firing of (a class of extended) transitions once they are enabled. Such interaction with the system environment is desired for systems such as reactive systems, workflow processes, or context-aware systems.

Definition 5.15 (Reactive Petri nets). *A reactive Petri net is defined as a 5-tuple $\mathcal{P} = \langle P, T_e, T_i, f, m_0\rangle$, where the set of transitions is split up into two disjoint sets of* external *and* internal *transitions $T = T_e \cup T_i$.*

The introduction of transitions that can fire automatically modifies the transition firing semantics as follows:

Definition 5.16. External transitions *(T_e) are fired according to the regular* may *fire semantics of Petri nets. That is, if a transition is enabled, it* may *fire. External transitions are fired as a consequence of an external input source.* Internal transitions *(T_i) are fired according to a* must *fire semantics. That is, if an internal transition is enabled, it must fire. Internal transitions process internal actions of the system.*

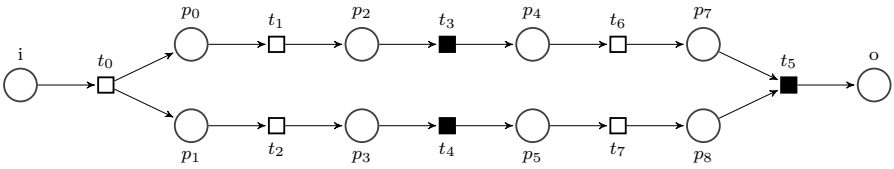


Figure 5.5: A Reactive Petri net [61].

Figure 5.5 shows an example of a reactive Petri net. In the figure, external transitions are represented by white squares, and internal transitions are represented by black ones. Transitions t_3, t_4 and t_5 are fired as soon as they become enabled, all other transitions are fired as a consequence of an action

in the environment. Whenever multiple internal transitions are simultaneously enabled they fire non-deterministically.

However, if both an internal and an external transitions are enabled at a certain marking, a conflict may arise. To avoid conflicts, internal transitions are required to fire with a higher priority than external transitions —that is, internal transitions will always fire before external transitions. This assumption corresponds to the perfect synchrony hypothesis [14] of reactive systems. The introduction of priorities in reactive Petri nets corresponds to the notion of static transition priorities of Section 5.2.1 —that is, reactive Petri nets are Petri nets with static priorities in which classes of transition priorities can fire reactively. A consequence of the introduction of transition priorities is that no external transition may fire until all enabled internal transitions have fired.

Definition 5.17 (**Net stability**). *A reactive Petri net, $\mathcal{P}$ is said to be stable at a marking m if and only if $\forall t_i \in T_i$, t_i is not enabled at m.*

Taken from Definition 5.17 we can deduce that a reactive Petri net can become unstable only by an external transition firing. In order to return to a stable reactive Petri net, enabled internal transitions must be fired. If after firing all enabled internal transitions a stable state is not reached, the system is said to *diverge*. Note that if a system diverges, that means that there must exists a loop where a set of internal transitions get infinitely enabled, and thus must fire. Otherwise it would be possible to reach a marking in which an external transition can fire as the result of an external state of the system.

The semantics of reactive Petri nets is equivalent to that of regular Petri nets if and only if the reactive Petri net satisfies the following constraints: (RC_1) internal and external transitions do not have conflicts. (RC_2) internal transitions with input places in common are free choice (i.e., can fire independently of each other), or there is no reachable marking enabling the two transitions.

RC_1 : $\forall t_i \in T_i, t_e \in T_e \ \bullet t_i \cap \bullet t_e = \phi$
RC_2 : $\forall t, t' \in T_i$ if $\bullet t \cap \bullet t' \neq \phi$, then either $\bullet t = \bullet t'$ or $\nexists m$ reachable marking, such that $m[t\rangle$ and $m[t'\rangle$ under the regular token game semantics

Conditions RC_1 and RC_2 are used to prove the equivalence between the semantics of regular Petri nets and the semantics of reactive Petri nets. Theorem 5.2 states that, for a given sequence of transition firings under the regular token game semantics of Petri nets, a permutation of the firing sequence reaches a stable state under the semantics of reactive Petri nets. The proof of the theorem can be found in Eshuis and Dehnert [61].

Theorem 5.2. *Given a reactive Petri net that satisfies constraints RC_1 and RC_2, suppose under the token games semantics, $\exists\, \Upsilon = t_0 \dots t_n$ a step and m, m' stable states such that $m[t_0\rangle m_1[t_1\rangle m_2 \cdots m_{n-1}[t_n\rangle m'$ and for $1 \leq i \leq n-1$ the markings m_i are unstable, written $m[\Upsilon\rangle m'$. Then, a permutation Υ' of Υ is such that $m[\Upsilon'\rangle m'$ under the reactive semantics.* □

5.2.3 Inhibitor Arcs

Inhibiting systems [142, 37] provide a means to explicitly express the absence of tokens in a Petri net place. Such conditions are introduced in Petri nets by adding *zero-testing* or *inhibitor* arcs. The extension of Petri nets with inhibitor arcs increases the computational power of Petri nets, making them equivalent to Turing machines [155].

Definition 5.18 (**Inhibiting Petri nets**). *A Petri net with inhibitor arcs is defined as 5-tuple* $\mathcal{P} = \langle P, T, f, f_\circ, m_0 \rangle$*, where the flow function* $f_\circ : P \times T \to \{0, 1\}$ *defines the inhibitor arcs. There can be maximum one inhibitor arc between a place and a transition.*

To account for inhibitor arcs the transition enabling rules are modified. For a given transition t, we make a distinction between the inputs coming from normal arcs ($\bullet t$), and the inputs coming from inhibitor arcs, $\circ t = \{p_{in} \in P \mid f_\circ(p_{in}, t) = 1\}$. Input places coming from inhibitor arcs are often referred to as the inhibiting places of a transition.

Definition 5.19. *Given a Petri net* $\mathcal{P}$ *with inhibitor arcs, a transition* $t \in T$ *is said to be enabled at a marking* m *if and only if,* $\forall\, p_{in} \in \bullet t,\ m(P_{in}) > f(p_{in}, t) \ \wedge\ \forall\, p_{\circ in} \in \circ t,\ m(p_{\circ in}) = 0$.

Figure 5.6 shows an example of a Petri net with one inhibitor arc such that $f_\circ(p_2, t_3) = 1$. Inhibitor arcs are decorated as circle-ended arcs ($\multimap$). In this Petri net, the enabling of transitions t_2 and t_3 depends on the marking of p_2. Only if p_2 is marked t_2 can be enabled; on the other hand, t_3 can be enabled only if p_2 is *not* marked. Because of this, t_2 is said to have priority over t_3.

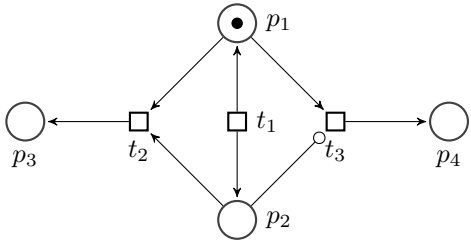


Figure 5.6: A Petri net with inhibitor arcs

Petri nets with inhibitor arcs are very useful for modeling systems for which the absence of resources needs to be explicitly expressed. Unfortunately, they also present the inconvenience that, in the general case, most of the Petri net properties (Section 5.1.1) become undecidable for Petri nets with inhibitor arcs. This is due to the equivalence of Petri nets with inhibitor arcs to Turing machines. Deciding if a certain marking is reachable in a Petri net with inhibitor arcs would be equivalent to deciding program termination for a Turing machine, which is undecidable.

Nonetheless, there are special cases in which it is possible to analyze the reachability problem for Petri nets with inhibitor arcs. Busi [26] studied *primitive systems*, a sub-class of Petri nets with inhibitor arcs for which it is possible to perform such analyses by means of reducing the primitive system to an equivalent Petri net without inhibitor arcs. This same kind of reduction could be applied to a general Petri net with place capacities and inhibitor arcs. Reinhardt [155] has also shown that the reachability problem is decidable for Petri net with inhibitor arcs if there exists an ordering of the places in the Petri net, such that a place only has an inhibitor arc to transitions which have an inhibitor arc from a preceding place.

5.2.4 Colored Petri Nets

Colored Petri nets (CPN) [99] are a Petri net extension aimed at combining Petri nets' description of concurrent processes with the definition of data types. The CPN extension maintains Petri net models simple and straightforward while it allows us to express more complex systems.

Definition 5.20 (Multiset). *A multiset g over a set L is a function $g : L \longrightarrow \mathbb{Z}^*$ assigning a weight (a non-negative integer) to each element of L.*

Definition 5.21 (Marking multiset). *Given sets A and L, a marking multiset $m : A \times L \longrightarrow \mathbb{Z}^*$, is defined as a multiset assigning a weight for each pair of elements (a, l) where $a \in A$ and $l \in L$.*

Definition 5.22 (Colored Petri nets). *Colored Petri nets are defined as a 5-tuple $\mathcal{P} = \langle P, T, f, \mathcal{L}, m_0 \rangle$, where P, T are defined as in regular Petri nets, $\mathcal{L}$ is the set of data types (colors) a place can hold, $f : (P \times T \times \mathcal{L}) \cup (T \times P \times \mathcal{L}) \longrightarrow \mathbb{Z}^*$ is a relation describing the flow over a multiset of tokens between two places by a transition firing, and m_0 is the initial marking multiset for places over the set of colors, $m_0 : P \times \mathcal{L} \longrightarrow \mathbb{Z}^*$.*

The flow of tokens through an arc defined by the flow function suggests that there can be as many arcs between a place and a transition as combinations of input token colors to output color tokens. Clearly, the definition of token colors requires a color-based definition of transition enabling.

Definition 5.23. *Given a CPN $\mathcal{P}$ and a marking multiset m, a transition $t \in T$ is said to be enabled for color $l \in \mathcal{L}$ if and only if $\forall\, p \in {\bullet t}$, $m(p, l) \geq f(p, t, l)$. If t is enabled for every color in $\mathcal{L}$, we simply say that t is enabled.*

Notation 5.2.

- *In order to facilitate the visualization of CPN, we label arcs by means of formulae expressing which colored tokens flow through the arc.*

Example 5.2. Figure 5.7 shows an example of a CPN describing the problem of the dining philosophers[4] (for a set of 4 philosophers *ph* and 4 chopsticks *cs*), where place think represents the thinking state of philosophers,

[4] `http://cpntools.org/documentation/examples/dining_philosophers`

place eat represents the eating state of philosophers, place free-chopsticks represents that chopsticks are free, transition take-chopsticks represents a state change from thinking to eating, and transition put-chopsticks represents a state change from eating to thinking. The color set for the Petri net is $\mathcal{L} = \{ph_0, \ldots, ph_3, cs_0, \ldots, cs_3\}$. In Figure 5.7 each philosopher is depicted by a *blue* colored token inscribed with the number of the philosopher (from 0 to 3), and each chopstick is depicted by a *orange* colored token inscribed with the number of the chopstick (from 0 to 3). In Figure 5.7, each place is sub-labeled with the accepted set of colors by the place (e.g., the set of the four philosophers in place think). Additionally, each arc is labeled with the token colors that flow through it when their adjacent transition is fired (e.g., cs_i, cs_{i+1} for the arc between place free-chopsticks and take-chopsticks).

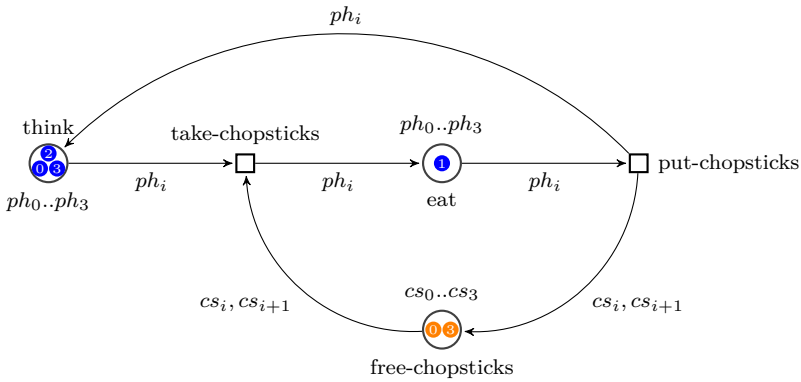


Figure 5.7: Colored Petri net for the 4 dining philosophers problem.

As an example execution of the Petri net, take the initial marking m_0 such that all philosophers are thinking and all chopsticks are free. For philosopher ph_1 to start eating, transition t_0 is fired taking the token of color ph_1 from place think and tokens of color cs_1, cs_2 from place free-chopsticks. The firing of this transition adds a token of color ph_1 to place eat as Figure 5.7 illustrates.

While CPNs allow to express complex systems with more ease than regular Petri nets, the two modeling approaches have the same expressive power —that is, it is always possible to get a regular Petri net from a CPN and *vice versa*. If the color set $\mathcal{L}$ is finite, the equivalent regular Petri net is also finite. If the color set $\mathcal{L}$ is infinite, the equivalent regular Petri net is infinite (i.e., it has an infinite set of places and transitions), we only deal with finite Petri nets in this dissertation (Definition 5.1).

Intuitively, the mapping from CPN to regular Petri nets takes every place p in the CPN and replaces it by a set of places p_l in the regular Petri net where $l \in \mathcal{L}$. A token of color l in place p in the CPN is represented by a regular token in place p_l in the regular Petri net. Every transition t in the CPN is replaced

by a set of transitions t_l in the regular Petri net. If there is an arc moving tokens of colors $l_1, \ldots, l_n$ from place p to transition t in the CPN, in the regular Petri net these arcs are represented by incoming arcs from place p_{l_k} to place t_l, where $k = 1, \ldots, n$. Similarly, an outgoing arc carries tokens of colors $l'_1, \ldots, l'_h$ from transition t to place p in the CPN, this is represented by several arcs from transition $t_{l'}$ to place $p_{l'_k}$ in the regular Petri net.

5.3 Conclusion

In this chapter we presented the generalities of the Petri net model, a mathematical model that effectively formalizes the structure and dynamics of the systems they represent. These two characteristics of Petri nets are key in the setting of Dynamically Adaptive Software Systems. Through the fine-grained definition of system states and actions, Petri nets provide a first hand view of the totality of the system structure. This characteristic of Petri nets is useful to define different system components and their relations soundly. Petri nets also provide a concrete view of the system dynamics and the live interaction between components by means of the token-game semantics. The Petri net marking represents the current state of the system at every step, and transition firing shows the information flow and component interaction as tokens move around.

Furthermore, modeling Dynamically Adaptive Software Systems using Petri nets allows us to reuse the analyses of system properties already existing for Petri nets. Petri nets are able to provide insightful information about future reachable states or possible system deadlocks. This characteristic of Petri nets is useful to aid the consistency management of dynamic adaptations in the system.

Taking these characteristics into consideration, it is safe to assert that Petri nets effectively satisfy the Requirements **M.1** to **M.4** defined in Chapter 2 for conflict resolution models.

However, on its own, the basic Petri net model does not exactly satisfy all requirements put forward for Dynamically Adaptive Software Systems in Chapter 2. Luckily enough, Petri net extensions exist that do address particular characteristics of systems that are not covered by the basic model. In this chapter we have only presented a couple of such extensions. It is important to notice that a vast variety of extensions exist. The extensions presented here were chosen under the light of Requirements **D.1**–**D.5** for Dynamically Adaptive Software Systems. We explain here which of these requirements are covered by the Petri net extensions:

Static priorities are used to establish an order between sets of transitions. Providing an order to the firing of sets of transitions can be used to separate the way in which system components interact. For example, static priorities can be used to separate the execution of environment-wide events to that of internal system events. In the case of COP, priorities can be

used to ensure that internal interactions between context objects are verified before any subsequent interaction with the surrounding execution environment takes place —that is, context activation and deactivation will have lower priority than triggering their internal interactions. Static priorities, thus, can be used to address Requirement **D.1**.

Reactive Petri nets are used to introduce reactivity to a system, allowing it to interact with other systems or system components. Providing a reactive semantics for Petri net transitions allows us to describe autonomic or proactive systems. Autonomy is a key requirement of Dynamically Adaptive Software Systems, which can adapt their behavior automatically as a consequence of changes in their surrounding execution environment. In the case of COP, the system must react to context activations and deactivations by adapting its behavior. Reactive Petri nets, thus, can be used to address Requirements **D.1** and **D.4**.

Inhibitor arcs are used to model systems where the state of a particular resource, absence or presence of it, dictates the kind of action the system can take. In the case of COP, testing if a context is active or not would be equivalent to testing for the absence of a resource. Using such a test it is possible to model different types of interaction between adaptations, allowing to activate or deactivate a particular context as long as another context is inactive. Inhibitor arcs, thus, can be used to address Requirement **D.5**.

Colored Petri nets are used to introduce the notion of types to the system, allowing us to deal with different types of events. In the case of COP the types introduced by CPN are used to differentiate between scoping mechanisms for for the system's behavioral adaptation. Scoping mechanisms can be used to adapt the system at different granularity levels such as object instances or full classes (Requirement **D**.2), or to isolate groups of behavioral adaptations that should not interact with each other under certain situations of the surrounding execution environment (Requirement **D**.3).

Chapter 6

Modeling and Managing Dynamically Adaptive Software Systems

In previous chapters we presented a specific class of Dynamically Adaptive Software Systems, and a model for the coordination and management of such software systems. We singled out context-aware systems as a highly dynamic class of software systems that allow us to adapt the system's behavior to changing situations at run time. Chapter 4 describes existing challenges in COP regarding the predictability of system behavior in the presence of run-time adaptations to the surrounding execution environment. Petri nets were presented in Chapter 5 as a model that captures the formal, structural and dynamic aspects of the software systems they represent. We provided the definition of the basic Petri nets model and different extensions that could be used for the coordination and management of behavioral adaptations to the surrounding execution environment.

The purpose of this chapter is to develop a formal basis for COP systems as a stepping stone for the development of a broader class of Dynamically Adaptive Software Systems such as the ones described in Section 3.1. The development of such a basis introduces context Petri nets [33], a Petri net-based formalism and programming model for COP systems.

The purpose of context Petri nets is threefold. First of all, we are interested in providing a formal and sound definition of COP systems, building up a basis for their development. Our proposed formal basis covers all aspects of COP, from the definition of contexts, their interactions and behavioral adaptations, to the selection of behavior, and composition of adaptations. This is achieved by using the structural definition of Petri nets and existing extensions thereof (i.e., inhibitor arcs and static priorities) which allow us to model the context states and their interactions. The benefit of having a common basis for COP is that it could facilitate the interoperability between different COP approaches, languages, or internal system modules. Additionally, a sound foundation of COP systems would facilitate their extensibility. Extensions can be proposed by modifying or building on top of developed tools and the initial formalization. Secondly, context Petri nets can help to ensure consistency of behavioral

adaptations at run-time as they are dynamically introduced to and withdrawn from the system. This is achieved by abstracting the state of adaptations in the system as the set of Petri net places, and providing a full representation of the adaptations' dynamicity as the set of Petri net transitions. Finally, taking advantage of the structural and formal definition of COP systems in terms of context Petri nets, we can reuse existing Petri net analysis techniques. Using such techniques, it is possible to analyze adaptations and their interactions at design time, with the objective of identifying possible inconsistencies between adaptations. This process is further explained in Chapter 7.

The next sections provide a definition of the model of context Petri nets presenting **(1)** their structure and formal definition Section 6.1, **(2)** their composition in order to represent a COP system Section 6.2, **(3)** their dynamic semantics to ensure a consistent interaction between adaptations Section 6.3, and **(4)** their programming interface with a COP language Section 6.4.

6.1 Context Petri Nets

In this section we provide the definition of context Petri nets (CoPNs) (read *co-pen*), as reactive Petri nets with inhibitor arcs and static transition priorities. We begin by providing the formal definition of CoPNs and a mapping of the main representative concepts of CoPN onto the corresponding concepts of COP languages.

Definition 6.1 (**Context Petri net**). *A **context Petri net (CoPN)** is defined as a 11-tuple $\mathcal{P} = \langle P_c, P_t, T_e, T_i, f, f_\circ, \rho, \mathcal{L}, m_0, \Sigma, \lambda \rangle$, where P_c is a finite set of context places, P_t is a finite set of temporary places, T_e is a finite set of external transitions, T_i is a finite set of internal transitions, f is the flow function defining regular arcs between places and transitions, $f_\circ$ is the flow function defining inhibitor arcs from places to transitions, ρ is a function defining transition priorities, $\mathcal{L}$ is a non-empty set of token colors, m_0 is the initial marking multiset of tokens to places, Σ is a non-empty finite set of labels, and λ is a function assigning labels to the places and transitions. The components of a CoPN are such that:*

(1) $P_c \cap P_t = \phi$
(2) $T_e \cap T_i = \phi$
(3) $(P_c \cup P_t) \cap (T_e \cup T_i) = \phi$
(4) $f : (P \times T \times \mathcal{L}) \cup (T \times P \times \mathcal{L}) \longrightarrow \mathbb{Z}^*$
(5) $f_\circ : P \times T \longrightarrow \{0, 1\}$
(6) $\rho : T \longrightarrow \mathbb{Z}^*$
(7) $\forall\, t \in T_e,\ \rho(t) = 0$
(8) $\forall\, t \in T_i,\ \rho(t) > 0$
(9) $\lambda : (P_c \cup P_t) \cup (T_e \cup T_i) \to \Sigma$
(10) $m_0 : P \times \mathcal{L} \longrightarrow \mathbb{Z}^*$

The components of a CoPN are characterized by: (1) The set of **context places** P_c, and the set of **temporary places** P_t are disjoint. At this point, the difference between context places and temporary places is syntactic. The usefulness of making such an explicit difference will become clear later in this section. (2) The set of **external transitions** T_e and the set of **internal**

transitions T_i are disjoint. (3) The sets of all places ($P_c \cup P_t$) and all transitions ($T_e \cup T_i$) are disjoint. (4) There cannot be arcs between two places or two transitions. Each arc defines how many tokens of each color flow from, or to places.[1] (5) There can be at most one inhibitor arc between a place and a transition. (6) Transitions are given a firing order priority. Higher priority transitions fire before lower priority ones, transitions with the same priority fire nondeterministically. (7) All external transitions have a priority of 0. (8) All internal transitions have a priority greater than 0. Transition priorities are defined in this way to comply with the semantics of reactive Petri nets —that is, internal transitions must fire before external transitions whenever they are enabled. (9) Every place and transition in the context Petri net is decorated with a label from Σ. (10) Finally, tokens are assigned to places by means of the initial marking multiset m_0.

Notation 6.1.

- *The set of all CoPNs is denoted as* $\mathscr{P}$.

Notation 6.2. *For a context* A, *the CoPN describing this context is given a set of labels* $\Sigma_{\mathtt{A}} = \{$A, *Pr(*A*), Pr(¬*A*), req(*A*), req(¬*A*), act(*A*), deac(*A*)*$\}$, *for its places and transitions, capturing the following intuition:*

- *External transitions, labeled req(*A*) or req(¬*A*), respectively represent a request for activating or deactivating context* A.
- *Internal transitions, labeled act(*A*) or deac(*A*), respectively represent the activation or deactivation of context* A.
- *The context place is labeled by the name of the context* A *it represents.*
- *Temporary places, labeled Pr(*A*) or Pr(¬*A*), respectively represent the states of preparing to activate or preparing to deactivate context* A.

In this dissertation we will use these labels of CoPN elements as a means to easily refer to them.

In general, a CoPN as described by Definition 6.1 is composed of many contexts, each described by a specific type of singleton CoPN as given in Definition 6.2.

Definition 6.2. *Given a context* A, *the* ***singleton CoPN*** *representing that context is defined as a unique CoPN structure* $\mathcal{C}_{\mathtt{A}} = \langle P_{c_A}, P_{t_A}, T_{e_A}, T_{i_A}, f_A, f_{\circ_A}, \rho_A, \mathcal{L}_A, m_{0_A}, \Sigma_A, \lambda_A \rangle$, *with:*

- *Label set* $\Sigma_A = \{$A, *Pr(*A*), Pr(¬*A*), req(*A*), req(¬*A*), act(*A*), deac(*A*)$\}$.
- *Context place* $P_{c_A} = \{p\}$ *where* $\lambda_A(p) =$ A.

[1] Even though in CoPN arcs carry exactly 1 token, here we provide a more general definition to enable the model to be easily extensible.

- *Temporary places* $P_{t_A} = \{p_1, p_2\}$ *where* $\lambda_A(p_1) = Pr(\mathtt{A})$ *and* $\lambda_A(p_2) = Pr(\neg\mathtt{A})$.
- *External transitions* $T_{e_A} = \{t_{e1}, t_{e2}\}$ *where* $\lambda_A(t_{e1}) = req(\mathtt{A})$ *and* $\lambda_A(t_{e2}) = req(\neg\mathtt{A})$.
- *Internal transitions* $T_{i_A} = \{t_{i1}, t_{i2}\}$ *where* $\lambda_A(t_{i1}) = act(\mathtt{A})$ *and* $\lambda_A(t_{i2}) = deac(\mathtt{A})$.
- *A finite set of colors* $\mathcal{L}_A = \{l_1, \ldots, l_n\}$.
- $\forall p \in P_{c_A} \cup P_{t_A}$ *and* $\forall t \in T_{e_A} \cup T_{i_A}$, $f_{\circ_A}(p, t) = 0$.
- ρ_A *is such that* $\forall\, t_e \in T_{e_A}$, $\rho(t_e) = 0$, *and* $\forall\, t_i \in T_{i_A}$, $\rho(t_i) = 2$. *The specific value of 2 is assigned to internal transitions to facilitate the extension of the model with another kind of internal transitions that have a lower priority with a value of 1 later in Section 9.3.2. Any other value greater than zero could be used to define the priority of internal transitions as given in Definition 6.1.*
- $\forall\, l \in \mathcal{L}_A$, f_A *is defined as:* $f_A(t_{e1}, p_1, l) = 1, f_A(p_1, t_{i1}, l) = 1, f_A(t_{i1}, p, l) = 1, f_A(p, t_{i2}, l) = 1, f_A(t_{e2}, p_2, l) = 1, f_A(p_2, t_{i2}, l) = 1$ *and 0 otherwise.*
- m_0 *an initial marking multiset of the places.*

Definition 6.2 defines the *structure* of a CoPN for a particular context A. The actual state of the context is represented by multiple instances of the CoPN according to its specific marking in a particular moment in time.

Notation 6.3.

- *We use* $\mathcal{C}$ *to denote a singleton CoPN, as defined in Definition 6.2.*
- *We use* $\mathscr{S}$ *to denote the set of all singleton CoPNs.*

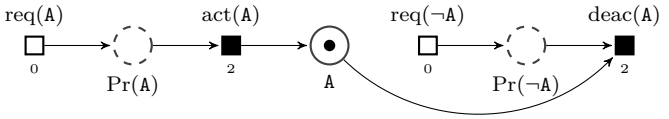


Figure 6.1: CoPN $\mathcal{P}_{\mathtt{A}}$ visual representations for a single context A.

Figure 6.1 shows the visual representation of the CoPN for a single context A. In the particular case of this figure, the initial marking multiset is given by $m_{0_A}(\mathtt{A}, black) = 1$ and $m_{0_A} = 0$ otherwise. Transition priorities are shown for each of the transitions in Figure 6.1. In the remainder of this dissertation, priorities are omitted from the figures, as they can be deduced from the color of the transitions: the darker the color the higher their priority. The following notation is assumed for the figures in the rest of this dissertation.

Notation 6.4.

- *External transitions are drawn as white squares* (□) *and have a priority of 0.*
- *Internal transitions are drawn as black squares* (■) *and have a priority of 2.*
- *Context places are drawn as circles with a solid edge* (◯).
- *Temporary places are drawn as circles with a dashed edge* (◌).
- *For a given* CoPN, $\mathcal{P} = \langle P_c, P_t, T_e, T_i, f, f_\circ, \rho, \mathcal{L}, m_0, \Sigma, \lambda \rangle$, *with* $|\mathcal{L}| = 1$, *the specification of the color is removed from the definition of flow functions and marking multisets. As a notation in this dissertation we will then use a black colored token, however any other color could be used.*

We now explain the correspondence between CoPN concepts and COP concepts. The mapping between concepts is defined with general CoPNs and COP systems in mind —that is, it takes into account systems where multiple interactive contexts are defined. However, to make the concepts clear we use the singleton CoPN shown in Figure 6.1 as a particular example.

Places in a CoPN are used to capture the possible states of contexts in the COP system.

- *Context places* represent a context and its state, such as the `NLBS` or `POSITIONING` contexts being active or inactive in the maps application example of Section 2.3.3.
- *Temporary places* express preparatory states for a context, facilitating consistency management and composition processes. For example, in Figure 6.1, the temporary places labeled $Pr(\texttt{A})$ or $Pr(\neg\texttt{A})$ are used to process requests to respectively activate or deactivate the context `A`. Later in Section 6.3.1 we show the need for introducing temporary places for managing the consistent activation of contexts.

Transitions represent actions that can be taken on the states of the system. In the case of CoPN, these actions correspond to context activations and deactivations. To avoid inconsistencies among different contexts defined in the system, activation and deactivation of contexts do not occur immediately. Context (de)activations need to be requested first and processed carefully, since the request may be denied if the activation or deactivation would violate constraints imposed by other contexts defined in the system.

- *External transitions* are used to *request* a context activation or deactivation in response to changes in the surrounding execution environment.
- *Internal transitions* deal with the constraints imposed by other contexts defined in the system (this will become clear in Section 6.2.2). Internal transitions trigger the actual activation or deactivation of contexts (i.e., they change the state of the context place).

Tokens represent context activations. The state of contexts is determined by the current marking of the CoPN. For example in Figure 6.1, context `A` is *active* if the place labeled `A` contains at least one token; *preparing for activation* if place labeled Pr(`A`) contains a token, *preparing for deactivation* if place labeled Pr($\neg$ `A`) contains a token. The number of tokens in a context place represent the activation count for that context, accounting for the interaction with other contexts defined in the system (Section 4.2).

Inhibitor arcs provide the possibility to verify the absence of tokens in a place. As we will se later, in CoPNs inhibitor arcs will be used to model interactions between contexts. For example, inhibitor arcs can be used to express that a context can be activated only if some other context is *not* active, as is the case for the exclusion dependency relation (Definition 6.13), or to express that a context must be deactivated if another context is *no longer* active, as in the case of the requirement dependency relation (Definition 6.16).

Up to this point we have provided a CoPN formalization that allows the definition of a single context and the possible actions on that context —that is, the activation and deactivation of a context through transition firings. Generally, a COP system is composed of multiple contexts —that is multiple instances of singleton CoPNs $\mathcal{C}$ such as the one shown in Figure 6.1. Thus, we need a means to compose multiple singleton CoPNs into a single CoPN with multiple contexts. The composition operator for CoPNs is defined in Section 6.2. Furthermore, context activations can depend on other contexts. Section 6.2.2 formalizes the semantics of interactions between contexts via context dependency relations. Section 6.3.1 then addresses the activation and deactivation interplay of contexts.

6.2 Building Context-Aware Systems

In the previous section we provided the formal definition of CoPNs and explained the intuition behind the mapping between CoPN and COP concepts. However, up to this point it is only possible to model independent contexts. In this section we provide the formal machinery to model more complex COP systems, composed of multiple contexts.

In a COP system, contexts can depend on each other. That is, a context activation can take place, or be refused, as a consequence of the activation, or activation state, of other contexts. The constraints that regulate such interactions are called **context dependency relations**. Intuitively, context dependency relations can be seen as a set of constraints on the state and dynamics (activation and deactivation) between contexts. Such constraints are expressed as rules that must be satisfied by a CoPN in order to respect the dependency relation. For example, to represent a context activation as a consequence of another context becoming active. Taking advantage of the fine-grained definition

of contexts provided by CoPNs, such an interaction can be modeled by connecting internal transitions of one context to the context places or temporary places of another context, via regular or inhibitor arcs. Each arc expresses (part of) a dependency constraint describing an interaction between two contexts.

Definition 6.3 (**Context dependency relations**). *A context dependency relation is defined as a tuple* $\langle R, \mathcal{C}_1, \ldots, \mathcal{C}_n \rangle$*, where* R *is a symbol representing the type of the relation defining the interaction between the singleton CoPNs* $\mathcal{C}_1, \ldots, \mathcal{C}_n$*. With each context dependency relation type* R *an extension function (*$\texttt{ext}_R$*) and constraining function (*$\texttt{cons}_R$*) is associated, describing the interaction between the contexts.*

Notation 6.5.

- *The set of all context dependency relations is denoted as* $\mathscr{R}$*.*

Definition 6.4. *For a context dependency relation* $\langle R, \mathcal{C}_1, \ldots, \mathcal{C}_n \rangle$ *the* $\texttt{ext}_R$ *extension function is* $\texttt{ext}_R : \mathscr{P} \times \mathscr{R} \rightarrow \mathscr{P}$ *and it is well defined for a given input pair* $(\mathcal{P}, R)$ *if and only if* $\mathcal{C}_i \subset \mathcal{P}$ *for* $1 \leq i \leq n$*.*

Definition 6.5. *For a context dependency relation* $\langle R, \mathcal{C}_1, \ldots, \mathcal{C}_n \rangle$ *the* $\texttt{cons}_R$ *constraining function is* $\texttt{cons}_R : \mathscr{P} \times \mathscr{R} \rightarrow \mathscr{P}$ *and it is well defined for a given input pair* $(\mathcal{P}, R)$ *if and only if* $\mathcal{C}_i \subset \mathcal{P}$ *for* $1 \leq i \leq n$*.*

The detailed definitions of the actual $\texttt{ext}_R$ and $\texttt{cons}_R$ function for the different types of context dependency relations are given in Section 6.2.2.

Definition 6.6. *The composition operator of CoPN is defined as a function* $\circ : \wp(\mathscr{S}) \times \wp(\mathscr{R}) \rightarrow \mathscr{P}$*, such that for a set of singleton CoPNs,* $\mathcal{S} \subseteq \mathscr{S}$ *and a set of dependency relations* $\mathcal{R} \subseteq \mathscr{R}$ *between those contexts in* $\mathcal{S}$*,* $\circ(\mathcal{S}, \mathcal{R}) = \texttt{cons}(\texttt{ext}(\texttt{union}(\mathcal{S}), \mathcal{R}), \mathcal{R})$*.*

Each of the auxiliary functions `union` (union), `ext` (extension) and `cons` (constraining) are explained in detail in the following subsections. Intuitively, the composition operator of CoPNs takes as input a set of contexts and context dependency relations between those contexts, and generates a CoPN composed of all those contexts (`union` function) and satisfying all context dependency relations. Context dependency relations are defined in a two-phase process. First, the `ext` function adds the places and transitions to the CoPN that are required to deal with specific cases of the interaction between the contexts (e.g., activate a context only if some other context is inactive). Second, the `cons` function adds additional arcs that may be required to deal with general cases of the interaction between contexts, (e.g., every deactivation of a context requests the deactivation of another context).

6.2.1 Unifying CoPNs

Let us start by presenting the `union` function, which allows us to obtain one single CoPN composed of a set of singleton CoPNs, following the ideas of place

fusion and transition fusion [130] of Petri nets. Given two CoPNs their union fuses together places and transitions that are equal in both CoPNs. Therefore, we first present the notion of equality for singleton CoPNs, places and transitions before presenting the definition of the `union` function.

Definition 6.7. *Given two places p, p' in a CoPN, they are said to be equal if and only if $\lambda(p) = \lambda(p')$.*

Definition 6.8. *Given two transitions t, t' in a CoPN, they are said to be equal if and only if:*

- $\lambda(t) = \lambda(t')$, *and*
- $\bullet t = \bullet t' \wedge \circ t = \circ t' \wedge t\bullet = t'\bullet$

Definition 6.9. *We define the equality between two singleton CoPNs, $\mathcal{C}_1 = \langle P_{c_1}, P_{t_1}, T_{e_1}, T_{i_1}, f_1, f_{\circ_1}, \rho_1, \mathcal{L}_1, m_{0_1}, \Sigma_1, \lambda_1\rangle$ and $\mathcal{C}_2 = \langle P_{c_2}, P_{t_2}, T_{e_2}, T_{i_2}, f_2, f_{\circ_2}, \rho_2, \mathcal{L}_2, m_{0_2}, \Sigma_2, \lambda_2\rangle$, if and only if $P_{c_1} = P_{c_2},\ P_{t_1} = P_{t_2},\ T_{e_1} = T_{e_2}$, and $T_{i_1} = T_{i_2}$.*

Definition 6.10 (Union). *The function `union`: $\wp(\mathscr{S}) \rightarrow \mathscr{P}$ is defined such that, for a given $\mathcal{S} = \{\mathcal{C}_1, \ldots, \mathcal{C}_n\} \subseteq \mathscr{S}$ finite set of singleton CoPNs, in which $\mathcal{C}_j = \langle P_{c_j}, P_{t_j}, T_{e_j}, T_{i_j}, f_j, f_{\circ_j}, \rho_j, \mathcal{L}_j, m_{0_j}, \Sigma_j, \lambda_j\rangle$, for $1 \leq j \leq n$, then `union`$(\mathcal{S}) = \mathcal{P}$, where $\mathcal{P} = \langle P_c, P_t, T_e, T_i, f, f_\circ, \rho, \mathcal{L}, m_0, \Sigma, \lambda\rangle$ is the CoPN defined by:*

$$P_c = \bigcup_{1\leq j\leq n} P_{c_j} \qquad P_t = \bigcup_{1\leq j\leq n} P_{t_j} \qquad T_e = \bigcup_{1\leq j\leq n} T_{e_j}$$

$$T_i = \bigcup_{1\leq j\leq n} T_{i_j} \qquad \mathcal{L} = \bigcup_{1\leq j\leq n} \mathcal{L}_j \qquad \Sigma = \bigcup_{1\leq j\leq n} \Sigma_j$$

$$m_0(p,l) = \max_{1\leq j\leq n} \{m_{0_j}(p,l)\}$$

$$\rho(t) = \rho_j(t) \quad \textit{if } t \in T_{e_j} \cup T_{i_j},\ \textit{for } 1 \leq j \leq n$$

$$\lambda(e) = \lambda_j(e) \quad \textit{if } e \in P_{c_j} \cup P_{t_j} \cup T_{e_j} \cup T_{i_j},\ \textit{for } 1 \leq j \leq n$$

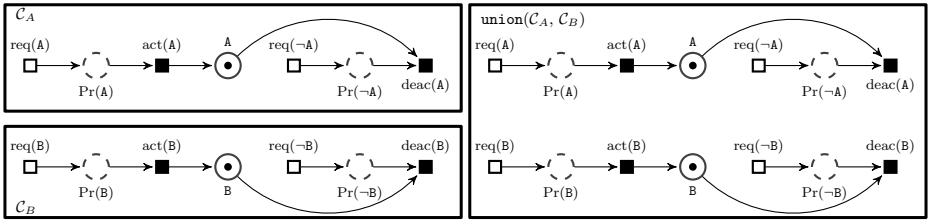
$$f(p,t,l) = \begin{cases} f_j(p,t,l) & \textit{if } (p,t,l) \in Dom(f_j) \textit{ for } 1 \leq j \leq n \\ 0 & \textit{otherwise} \end{cases}$$

$$f(t,p,l) = \begin{cases} f_j(t,p,l) & \textit{if } (p,t,l) \in Dom(f_j) \textit{ for } 1 \leq j \leq n \\ 0 & \textit{otherwise} \end{cases}$$

$$f_\circ(p,t) = \begin{cases} f_{\circ_j}(p,t) & \textit{if } (p,t,l) \in Dom(f_j) \textit{ for } 1 \leq j \leq n \\ 0 & \textit{otherwise} \end{cases}$$

It is possible to unify different singleton CoPNs which represent one same context. If this is the case, the two CoPNs are merged together into one singleton CoPN by means of place and transition fusion. Places are fused whenever they have the same label, and transitions are fused whenever they are equal according to Definition 6.8. To ensure that interactions between contexts are not compromised as places are fused, the initial marking of the unified CoPN is defined as the maximum of the respective initial markings for each singleton CoPN. All the situations for which the context was activated are preserved by taking the maximum number of tokens between the initial singleton CoPNs. Example 6.2, later in this chapter, illustrates this choice of the marking multiset when taking the union of CoPNs. This choice allows for a correct interaction of contexts and their multiple activations.

Example 6.1. As an example of the `union` function, let us take two singleton CoPNs $\mathcal{C}_A = \langle P_{c_A}, P_{t_A}, T_{e_A}, T_{i_A}, f_A, f_{\circ_A}, \rho_A, \mathcal{L}_A, m_{0_A}, \Sigma_A, \lambda_A \rangle$ and $\mathcal{C}_B = \langle P_{c_B}, P_{t_B}, T_{e_B}, T_{i_B}, f_B, f_{\circ_B}, \rho_B, \mathcal{L}_B, m_{0_B}, \Sigma_B, \lambda_B \rangle$ as shown in Figure 6.2a. Figure 6.2b shows the CoPN representing their union, $\texttt{union}(\mathcal{C}_A, \mathcal{C}_B)$.



(a) Two singleton CoPNs.

(b) A CoPN generated by the union of the two singleton CoPNs on the left.

Figure 6.2: Application of the `union` function to two singleton CoPNs.

6.2.2 Extending and Constraining CoPNs

In this section we investigate how to define interactions between contexts. As mentioned in Definition 6.3, context dependency relations impose constraints on the way contexts should and should not interact with each other. A context activation can take place, or be refused, as a consequence of the activation of other contexts. Context dependency relations extend (`ext`) and constrain (`cons`) a CoPN by adding extra places, transitions, and arcs. Such added elements are used, for example, to request the activation of a context every time another context is activated (e.g., in the case of a causality dependency relation). Such dependencies are expressed by connecting internal transitions of one context to the context places or temporary places of another context, via regular or inhibitor arcs.

Definition 6.11 (**Extension**). *The function* $\texttt{ext}$: $\mathscr{P} \times \wp(\mathscr{R}) \rightarrow \mathscr{P}$ *is defined such that, for a* CoPN $\mathcal{P} = \langle P_c, P_t, T_e, T_i, f, f_\circ, \rho, \mathcal{L}, m_0, \Sigma, \lambda\rangle \in \mathscr{P}$ *and* $\mathcal{R} = \{\langle R_1, \mathcal{C}_{1_1}, \ldots, \mathcal{C}_{q_1}\rangle, \ldots, \langle R_n, \mathcal{C}_{1_n}, \ldots, \mathcal{C}_{q_n}\rangle\} \subseteq \mathscr{R}$ *a finite set of context dependency relations over the singleton* CoPNs *in* $\mathcal{P}$,
$\texttt{ext}(\mathcal{P}, \mathcal{R}) = \texttt{ext}_{R_1}(\texttt{ext}_{R_2}(\texttt{ext}_{R_3}(\ldots), \langle R_2, \mathcal{C}_{1_2}, \ldots, \mathcal{C}_{q_2}\rangle), \langle R_1, \mathcal{C}_{1_1}, \ldots, \mathcal{C}_{q_1}\rangle)$.

The definitions of the $\texttt{ext}_R$ functions specific to each of the context dependency relations currently defined in CoPNs are given in Definitions 6.13 through 6.17. Theorem 6.1 demonstrates that the order in which the $\texttt{ext}_R$ functions are applied does not affect the result of applying the function.

Definition 6.12 (**Constraining**). *The function* $\texttt{cons}$: $\mathscr{P} \times \wp(\mathscr{R}) \rightarrow \mathscr{P}$ *is defined such that, for a* CoPN $\mathcal{P} = \langle P_c, P_t, T_e, T_i, f, f_\circ, \rho, \mathcal{L}, m_0, \Sigma, \lambda\rangle \in \mathscr{P}$ *and* $\mathcal{R} = \{\langle R_1, \mathcal{C}_{1_1}, \ldots, \mathcal{C}_{q_1}\rangle, \ldots, \langle R_n, \mathcal{C}_{1_n}, \ldots, \mathcal{C}_{q_n}\rangle\} \subseteq \mathscr{R}$ *a finite set of context dependency relations over the singleton* CoPNs *in* $\mathcal{P}$,
$\texttt{cons}(\mathcal{P}, \mathcal{R}) = \texttt{cons}_{R_1}(\texttt{cons}_{R_2}(\texttt{cons}_{R_3}(\ldots), \langle R_2, \mathcal{C}_{1_2}, \ldots, \mathcal{C}_{q_2}\rangle), \langle R_1, \mathcal{C}_{1_1}, \ldots, \mathcal{C}_{q_1}\rangle)$.

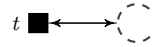
The definitions of the $\texttt{cons}_R$ functions specific to each of the context dependency relations currently defined in CoPNs are given in Definitions 6.13 through 6.17. Theorem 6.1 demonstrates that the order in which the $\texttt{cons}_R$ functions are applied does not affect the result of applying the function.

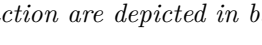
The CoPN model currently supports seven types of context dependency relations, among which the four types of context dependency relations initially defined in Subjective-C [77]: *exclusion*, *causality* (called weak inclusion in Subjective-C), *implication* (called strong inclusion in Subjective-C), and *requirement*. An additional *conjunction* dependency relation is also added (this type of relation has been informally introduced in Ambience [75] and more recently has also been formally added as part of EventCJ [104, 105]). We argue that other context dependency relations could be defined in a similar fashion. Section 9.3 is a case in point by introducing two context dependency relations, *suggestion* and *disjunction*. We now define the initial five context dependency relations and their associated `ext` and `cons` functions. For each context dependency relation we also provide a visual representation of the corresponding CoPN. The maps application is used as an example to demonstrate the interaction between contexts for each context dependency relation.

Notation 6.6.

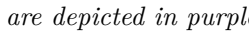
- *CoPNs currently support 5 types of context dependency relations* $\{E, C, I, Q, \wedge\}$, *for the exclusion, causality, implication, requirement, and conjunction dependency relations.*

Notation 6.7. *The following notation is used for the visualization of* CoPNs, $\mathcal{P} = \langle P_c, P_t, T_e, T_i, f, f_\circ, \rho, \mathcal{L}, m_0, \Sigma, \lambda\rangle$*:*

- *Given a place* $p \in P_c \cup P_t$, *a transition* $t \in T_i$, *and two arcs* $f(p, t, l)$ *and* $f(t, p, l)$, *visually these arcs are represented as a double arrow arc (e.g.,*  *).*

- *For every context dependency relation, the elements added to the CoPN by the* **ext** *function are depicted in blue (e.g.,* 

).

- *For every context dependency relation, the arcs added to the CoPN by the* **cons** *function are depicted in purple (e.g.,* 

).

- *To simplify the conditions of the formulae defining context dependency relations in what follows we refer to specific places and transitions by their labels, for example, the formula "$p_t \in t\bullet$ where $\lambda(p_t) = Pr(\mathtt{A})$" is simplified as "$Pr(\mathtt{A}) \in t\bullet$".*

Exclusion dependency relation

An *exclusion* dependency relation between two contexts A and B represents a restriction such that the two contexts cannot be active at the same time. However, both contexts may be simultaneously inactive.

Definition 6.13 (Exclusion). *The exclusion dependency relation between two contexts (*$\mathtt{A}\square\!\!-\!\!\square\mathtt{B}$*) is defined as the tuple $\langle E, \mathcal{C}_A, \mathcal{C}_B\rangle$, where $\mathcal{C}_A$ and $\mathcal{C}_B$ are two different singleton CoPNs. The composed CoPN defining an exclusion dependency relation, $\mathcal{P} = \langle P_c, P_t, T_e, T_i, f, f_\circ, \rho, \mathcal{L}, m_0, \Sigma, \lambda\rangle$ is obtained by the union of the singleton CoPNs, $\mathcal{P} = \mathbf{union}(\{\mathcal{C}_A, \mathcal{C}_B\})$, and the application of the $\mathbf{ext}_E$ and $\mathbf{cons}_E$ functions to $\mathcal{P}$.*

The $\mathbf{ext}_E$ function is defined as $\mathbf{ext}_E(\mathcal{P}, \langle E, \mathcal{C}_A, \mathcal{C}_B\rangle) = \mathcal{P}$. That is, this function does not modify the definition of the CoPN $\mathcal{P}$.

This is because, from the intended interaction defined by the exclusion dependency relation, there is no particular case to take into account for the interaction between the two contexts involved in the dependency relation, thus there is no need to add any elements with the $\mathbf{ext}_E$ function. The interaction only expresses that every time a context is to be activated, it must be ensured that the other context is not active; such a case is dealt with by the $\mathbf{cons}_E$ function.

The $\mathbf{cons}_E$ function is defined as $\mathbf{cons}_E(\mathcal{P}, \langle E, \mathcal{C}_A, \mathcal{C}_B\rangle) = \mathcal{P}'$, between two singleton CoPNs $\mathcal{C}_A = \langle P_{c_A}, P_{t_A}, T_{e_A}, T_{i_A}, f_A, f_{\circ_A}, \rho_A, \mathcal{L}_A, m_{0_A}, \Sigma_A, \lambda_A\rangle$ and $\mathcal{C}_B = \langle P_{c_B}, P_{t_B}, T_{e_B}, T_{i_B}, f_B, f_{\circ_B}, \rho_B, \mathcal{L}_B, m_{0_B}, \Sigma_B, \lambda_B\rangle$, where $\mathcal{C}_A, \mathcal{C}_B \subset \mathcal{P} = \langle P_c, P_t, T_e, T_i, f, f_\circ, \rho, \mathcal{L}, m_0, \Sigma, \lambda\rangle$ and $\mathcal{P}' = \langle P_c, P_t, T_e, T_i, f, f'_\circ, \rho, \mathcal{L}, m_0, \Sigma, \lambda\rangle$, such that

$$f'_\circ(p,t) = \begin{cases} 1 & \text{if } \lambda(p) = \mathtt{B} \wedge \mathtt{A} \in t\bullet \wedge \mathtt{A} \notin \bullet t \quad (6.1) \\ 1 & \text{if } \lambda(p) = \mathtt{A} \wedge \mathtt{B} \in t\bullet \wedge \mathtt{B} \notin \bullet t \quad (6.2) \\ f_\circ(p,t) & \text{otherwise} \quad (6.3) \end{cases}$$

The inhibitor arcs introduced by $\mathbf{cons}_E$ represent the condition that only one of the contexts can be active at a time. Transitions activating a context are enabled if and only if the other context is inactive (Equations (6.1) and (6.2)).

Figure 6.3 illustrates the CoPN representing the exclusion dependency relation $\langle E, \mathcal{C}_{\text{PRIV}}, \mathcal{C}_{\text{POS}} \rangle$ between the PRIVATE (PRIV) and POSITIONING (POS) contexts of the maps application (Section 2.3.3).

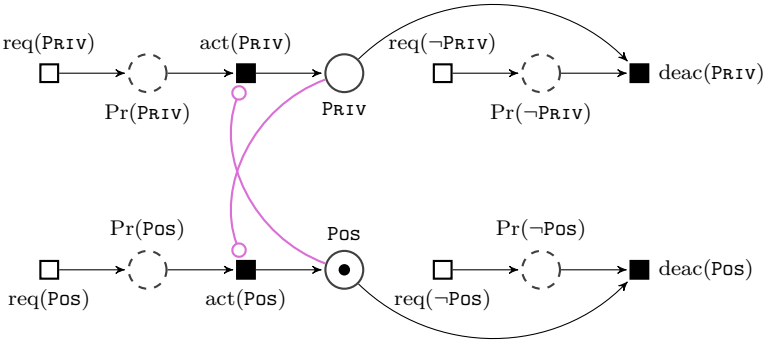


Figure 6.3: Exclusion dependency relation (PRIV□–□POS).

Exclusion dependency relations are used to model situations that should not occur simultaneously. In the maps application, the PRIVATE and POSITIONING contexts should not be active at the same time, given their conflicting behavior (concealing user information and broadcasting the user's position, respectively). Hence, an exclusion dependency relation is defined between the two contexts, as Figure 6.3 illustrates. The consequence of such a relation is that the exchange between the PRIVATE and POSITIONING contexts must occur by first deactivating the currently active context and then activating the other one.

Causality dependency relation

A *causality* dependency relation between two contexts A and B represents the situation in which the (de)activation of the source context A automatically triggers the (de)activation of the target context B. However, (de)activation of the target context B is independent from the source context.

Definition 6.14 (Causality). *The causality dependency relation between two contexts (A–▷B) is defined as the tuple $\langle C, \mathcal{C}_A, \mathcal{C}_B \rangle$, where $\mathcal{C}_A$ and $\mathcal{C}_B$ are two different singleton CoPNs. The composed CoPN defining a causality dependency relation, $\mathcal{P} = \langle P_c, P_t, T_e, T_i, f, f_\circ, \rho, \mathcal{L}, m_0, \Sigma, \lambda \rangle$ is obtained by the union of the singleton CoPNs, $\mathcal{P} = \textbf{\textit{union}}(\{\mathcal{C}_A, \mathcal{C}_B\})$, and the application of the $\textbf{\textit{ext}}_C$ and $\textbf{\textit{cons}}_C$ functions to $\mathcal{P}$.*

The $\textbf{\textit{ext}}_C$ function is defined as $\textbf{\textit{ext}}_C(\mathcal{P}, \langle C, \mathcal{C}_A, \mathcal{C}_B \rangle) = \mathcal{P}'$, between two singleton CoPNs $\mathcal{C}_A = \langle P_{c_A}, P_{t_A}, T_{e_A}, T_{i_A}, f_A, f_{\circ_A}, \rho_A, \mathcal{L}_A, m_{0_A}, \Sigma_A, \lambda_A \rangle$ and $\mathcal{C}_B = \langle P_{c_B}, P_{t_B}, T_{e_B}, T_{i_B}, f_B, f_{\circ_B}, \rho_B, \mathcal{L}_B, m_{0_B}, \Sigma_B, \lambda_B \rangle$, where $\mathcal{C}_A, \mathcal{C}_B \subset \mathcal{P} = \langle P_c, P_t, T_e, T_i, f, f_\circ, \rho, \mathcal{L}, m_0, \Sigma, \lambda \rangle$ and $\mathcal{P}' = \langle P_c, P_t, T_e, T_i', f', f_\circ', \rho', \mathcal{L}, m_0, \Sigma, \lambda' \rangle$, such that:

$$T_i' = T_i \cup \{t'\}$$

$$\lambda'(e) = \begin{cases} \lambda(e) & \text{if } e \in P_c \cup P_t \cup T_e \cup T_i \\ deac(\mathtt{A}) & \text{if } e = t' \end{cases}$$

$$\rho'(t) = \begin{cases} \rho(t) & \text{if } t \in T_e \cup T_i \\ 2 & \text{if } t = t' \end{cases}$$

$$f'(t,p,l) = \begin{cases} f(t,p,l) & \text{if } t \in T_e \cup T_i \ \wedge \ p \in P_c \cup P_t \ \wedge \ l \in \mathcal{L} \\ 0 & \text{otherwise} \end{cases}$$

$$f'(p,t,l) = \begin{cases} f(p,t,l) & \text{if } p \in P_c \cup P_t \ \wedge \ t \in T_e \cup T_i \ \wedge \ l \in \mathcal{L} \\ 1 & \text{if } \lambda(p) = \mathtt{A} \ \wedge \ t = t' \ \wedge \ l \in \mathcal{L} \\ 1 & \text{if } \lambda(p) = Pr(\neg\mathtt{A}) \ \wedge \ t = t' \ \wedge \ l \in \mathcal{L} \\ 0 & \text{otherwise} \end{cases}$$

$$f'_\circ(p,t) = \begin{cases} f_\circ(p,t) & \text{if } p \in P_c \cup P_t \ \wedge \ t \in T_e \cup T_i \\ 1 & \text{if } \lambda(p) = \mathtt{B} \wedge t = t' \\ 0 & \text{otherwise} \end{cases}$$

The transition introduced by the $\mathtt{ext}_C$ *function manages the interaction for the deactivation of the source context whenever the target context is inactive. This case can arise through the deactivation of the target context, since it is independent of the source context.*

The $\mathtt{cons}_C$ *function is defined as* $\mathtt{cons}_C(\mathcal{P}, \langle C, \mathcal{C}_A, \mathcal{C}_B \rangle) = \mathcal{P}'$*, between two singleton CoPNs* $\mathcal{C}_A = \langle P_{c_A}, P_{t_A}, T_{e_A}, T_{i_A}, f_A, f_{\circ_A}, \rho_A, \mathcal{L}_A, m_{0_A}, \Sigma_A, \lambda_A \rangle$ *and* $\mathcal{C}_B = \langle P_{c_B}, P_{t_B}, T_{e_B}, T_{i_B}, f_B, f_{\circ_B}, \rho_B, \mathcal{L}_B, m_{0_B}, \Sigma_B, \lambda_B \rangle$*, where* $\mathcal{C}_A, \mathcal{C}_B \subset \mathcal{P} = \langle P_c, P_t, T_e, T_i, f, f_\circ, \rho, \mathcal{L}, m_0, \Sigma, \lambda \rangle$ *and* $\mathcal{P}' = \langle P_c, P_t, T_e, T_i, f', f_\circ, \rho, \mathcal{L}, m_0, \Sigma, \lambda \rangle$*, such that:*

$$f'(p,t,l) = \begin{cases} 1 & \text{if } \lambda(p) = \mathtt{B} \ \wedge \ \mathtt{A} \in {\bullet t} \ \wedge \ \mathtt{A} \notin t\bullet \ \wedge \\ & \mathtt{B} \notin {\circ t} \ \wedge \ l \in \mathcal{L} \qquad (6.4) \\ f(p,t,l) & \text{otherwise} \qquad (6.5) \end{cases}$$

$$f'(t,p,l) = \begin{cases} 1 & \text{if } \lambda(p) = \mathtt{B} \ \wedge \ \mathtt{A} \in {\bullet t} \ \wedge \ \mathtt{A} \notin t\bullet \ \wedge \\ & \mathtt{B} \notin {\circ t} \ \wedge \ l \in \mathcal{L} \qquad (6.6) \\ 1 & \text{if } \lambda(p) = Pr(\neg\mathtt{B}) \ \wedge \ \mathtt{A} \in {\bullet t} \ \wedge \ \mathtt{A} \notin t\bullet \ \wedge \\ & \mathtt{B} \notin {\circ t} \ \wedge \ l \in \mathcal{L} \qquad (6.7) \\ 1 & \text{if } \lambda(p) = Pr(\mathtt{B}) \ \wedge \ \mathtt{A} \in t\bullet \ \wedge \mathtt{A} \notin {\bullet t} \ \wedge \ l \in \mathcal{L} \qquad (6.8) \\ f(p,t,l) & \text{otherwise} \qquad (6.9) \end{cases}$$

The arcs introduced by the $\mathtt{cons}_C$ *function are used to forward the deactivation of the source context to the target context, whenever the source context*

is deactivated and the target context is active, cf. Equations (6.4), (6.6), and (6.7). The behavior provided by these constraints complements the deactivation rule for the source context introduced by the $\mathbf{ext}_C$ *function. Additionally, every activation of the source context requests the activation of the target context, given that the source context is not an input place for such transition, cf. Equation* (6.8).

Figure 6.4 illustrates the CoPN representing the causality dependency relation $\langle C, \mathcal{C}_{\mathtt{W}}, \mathcal{C}_{\mathtt{C}} \rangle$ between the `WLAN (W)` and `CONNECTIVITY (C)` contexts of the maps application (Section 2.3.3).

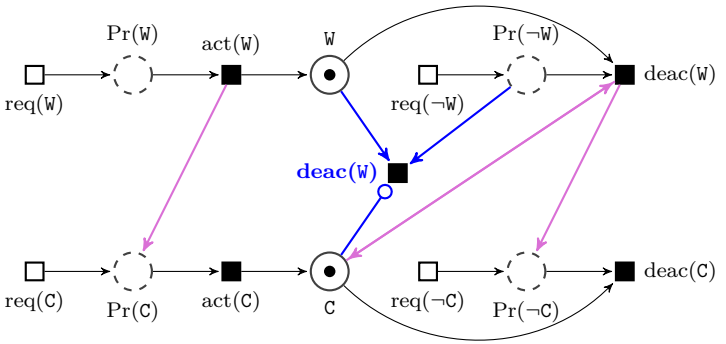


Figure 6.4: Causality dependency relation (W–▷C).

Implication dependency relation

An *implication* dependency relation between two contexts `A` and `B`, similarly to the causality dependency relation, represents a situation in which the (de)activation of the source context `A` automatically triggers the (de)activation of the target context `B`. The relation is said to be an implication because it follows the semantics of the implication logic operator ($\Rightarrow$) by ensuring that whenever context `B` is inactive, context `A` will automatically become inactive. Note that as long as context `B` is not made inactive, its deactivation is independent from that of context `A`.

Definition 6.15 (Implication). *The implication dependency relation between two contexts (*A–▶B*) is defined as the tuple* $\langle I, \mathcal{C}_A, \mathcal{C}_B \rangle$*, where* $\mathcal{C}_A$ *and* $\mathcal{C}_B$ *are two different singleton CoPNs. The composed CoPN defining an implication dependency relation,* $\mathcal{P} = \langle P_c, P_t, T_e, T_i, f, f_\circ, \rho, \mathcal{L}, m_0, \Sigma, \lambda \rangle$ *is obtained by the union of each of the singleton CoPNs,* $\mathcal{P} = \mathbf{union}(\{\mathcal{C}_A, \mathcal{C}_B\})$*, and the application of the* $\mathbf{ext}_I$ *and* $\mathbf{cons}_I$ *functions to* $\mathcal{P}$*.*

The $\mathbf{ext}_I$ *function is defined as* $\mathbf{ext}_I(\mathcal{P}, \langle I, \mathcal{C}_A, \mathcal{C}_B \rangle) = \mathcal{P}'$*, between two singleton CoPNs* $\mathcal{C}_A = \langle P_{c_A}, P_{t_A}, T_{e_A}, T_{i_A}, f_A, f_{\circ_A}, \rho_A, \mathcal{L}_A, m_{0_A}, \Sigma_A, \lambda_A \rangle$ *and* $\mathcal{C}_B = \langle P_{c_B}, P_{t_B}, T_{e_B}, T_{i_B}, f_B, f_{\circ_B}, \rho_B, \mathcal{L}_B, m_{0_B}, \Sigma_B, \lambda_B \rangle$*, where* $\mathcal{C}_A, \mathcal{C}_B \subset \mathcal{P} =$

$\langle P_c, P_t, T_e, T_i, f, f_\circ, \rho, \mathcal{L}, m_0, \Sigma, \lambda\rangle$ *and* $\mathcal{P}' = \langle P_c, P_t, T_e, T_i', f', f_\circ', \rho', \mathcal{L}, m_0, \Sigma, \lambda'\rangle$, *such that:*

$$T_i' = T_i \cup \{t'\}$$

$$\lambda'(e) = \begin{cases} \lambda(e) & \text{if } e \in P_c \cup P_t \cup T_e \cup T_i \\ deac(\mathtt{A}) & \text{if } e = t' \end{cases}$$

$$\rho'(t) = \begin{cases} \rho(t) & \text{if } t \in T_e \cup T_i \\ 2 & \text{if } t = t' \end{cases}$$

$$f'(t,p,l) = \begin{cases} f(t,p,l) & \text{if } t \in T_e \cup T_i \wedge p \in P_c \cup P_t \wedge l \in \mathcal{L} \\ 1 & \text{if } t = t' \wedge \lambda(p) = \mathtt{A} \wedge l \in \mathcal{L} \\ 1 & \text{if } t = t' \wedge \lambda(p) = Pr(\neg\mathtt{A}) \wedge l \in \mathcal{L} \\ 0 & \text{otherwise} \end{cases}$$

$$f'(p,t,l) = \begin{cases} f(p,t,l) & \text{if } p \in P_c \cup P_t \wedge t \in T_e \cup T_i \wedge l \in \mathcal{L} \\ 1 & \text{if } \lambda(p) = \mathtt{A} \wedge t = t' \wedge l \in \mathcal{L} \\ 0 & \text{otherwise} \end{cases}$$

$$f_\circ'(p,t) = \begin{cases} f_\circ(p,t) & \text{if } p \in P_c \cup P_t \wedge t \in T_e \cup T_i \\ 1 & \text{if } \lambda(p) = \mathtt{B} \wedge t = t' \\ 1 & \text{if } \lambda(p) = Pr(\mathtt{B}) \wedge t = t' \\ 1 & \text{if } \lambda(p) = Pr(\neg\mathtt{A}) \wedge t = t' \\ 0 & \text{otherwise} \end{cases}$$

As for the causality dependency relation, the $\mathbf{ext}_I$ *function also introduces a deactivation transition,* t' *for the source context of the dependency relation. This transition manages the interaction case in which whenever the target contexts becomes inactive, the source context is requested for deactivation as as long as it remains active. This transition is only enabled if the target context is not preparing to activate, to avoid deactivating the source context if the target context will become active.*

The $\mathbf{cons}_I$ *function is defined as* $\mathbf{cons}_I(\mathcal{P}, \langle I, \mathcal{C}_A, \mathcal{C}_B\rangle) = \mathcal{P}'$, *between two singleton CoPNs* $\mathcal{C}_A = \langle P_{c_A}, P_{t_A}, T_{e_A}, T_{i_A}, f_A, f_{\circ_A}, \rho_A, \mathcal{L}_A, m_{0_A}, \Sigma_A, \lambda_A\rangle$ *and* $\mathcal{C}_B = \langle P_{c_B}, P_{t_B}, T_{e_B}, T_{i_B}, f_B, f_{\circ_B}, \rho_B, \mathcal{L}_B, m_{0_B}, \Sigma_B, \lambda_B\rangle$, *where* $\mathcal{C}_A, \mathcal{C}_B \subset \mathcal{P} = \langle P_c, P_t, T_e, T_i, f, f_\circ, \rho, \mathcal{L}, m_0, \Sigma, \lambda\rangle$ *and* $\mathcal{P}' = \langle P_c, P_t, T_e, T_i, f', f_\circ, \rho, \mathcal{L}, m_0, \Sigma, \lambda\rangle$, *such that:*
$\forall\, p \in P_c \cup P_t$, $\forall\, t \in T_e \cup T_i$ *and* $\forall\, l \in \mathcal{L}$

$$f'(p,t,l) = f(p,t,l)$$

$$f'(t,p,l) = \begin{cases} 1 & \text{if } \lambda(p) = Pr(\mathtt{B}) \wedge \mathtt{A} \in t\bullet \wedge \mathtt{A} \notin \bullet t \\ & \wedge\, l \in \mathcal{L} \qquad (6.10) \\ 1 & \text{if } \lambda(p) = Pr(\neg\mathtt{B}) \wedge \mathtt{A} \in \bullet t \wedge \mathtt{A} \notin t\bullet \\ & \wedge\, \mathtt{B} \notin \circ t \wedge l \in \mathcal{L} \qquad (6.11) \\ f(t,p,l) & \text{otherwise} \qquad (6.12) \end{cases}$$

The arcs introduced by $\mathtt{cons}_I$ *define that for every activation of the source context* A*, for which* A *is not an input, the target context* B *is requested for activation, cf. Equation* (6.10)*, and for every deactivation of* A *for which* B *is not an inhibitor,* B *is requested for deactivation, cf. Equation* (6.11)*.*

Figure 6.5 illustrates the CoPN representing the implication dependency relation $\langle I, \mathcal{C}_{\mathtt{N}}, \mathcal{C}_{\mathtt{Pos}} \rangle$ between the NLBS (N) and POSITIONING (Pos) contexts of the maps application (Section 2.3.3).

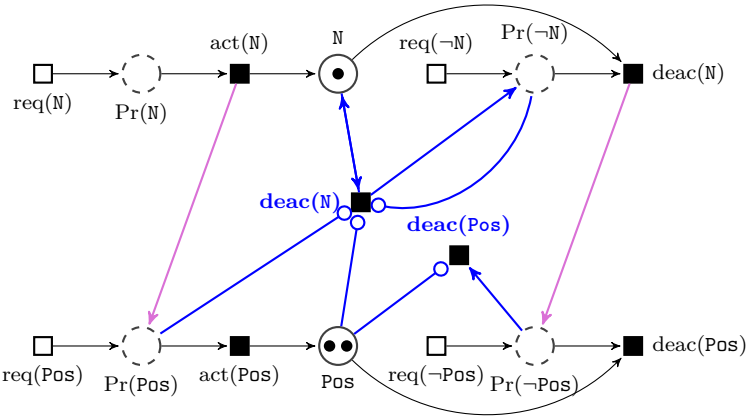


Figure 6.5: Implication dependency relation (N–▶Pos).

The formal definition of the implication dependency relation solves the *deactivation cycle* and *accidental interaction* problems explained in Section 4.4.3. The loop between the deactivation of the source context and the deactivation of the target context no longer exists because, deactivations of the target context no longer request the deactivation of the source context. This was an imprecision in the informal description given to the context dependency relation (Table 4.5). The accidental interaction no longer exists because deactivation requests of the target context are not forwarded to the source context. The transition introduced by the $\mathtt{ext}_I$ function ensures that the source context becomes inactive whenever the target context is inactive and it is not preparing to activate (this constraint is necessary to ensure that the target context can be activated).

Implication dependency relations can be used, for example, in situations where a containment interaction exists between contexts in the real world, or in situations where services provided by one context can be used by another context. The maps application can retrieve the user position based on different available services. As NBLS provides services that are explicitly used by context POSITIONING, every time the former context is activated, it can be ensured that the services of the latter context remain available. Conversely, if we are no longer in the POSITIONING context, the services of NBLS are no longer needed and thus can be turned off, for example, to save battery life.

Requirement dependency relation

A *requirement* dependency relation between two contexts A and B represents the situation in which the activation of the source context A is possible only if the target context B is already active. The deactivation of the two contexts can occur independently. However, if the target context B becomes inactive, it must be ensured that the source context A becomes inactive as well.

Definition 6.16 (Requirement). *The requirement dependency relation between two contexts (*A$\blacktriangleleft$B*) is defined as a tuple* $\langle Q, \mathcal{C}_A, \mathcal{C}_B \rangle$*, where* $\mathcal{C}_A$ *and* $\mathcal{C}_B$ *are two different singleton CoPNs. The composed CoPN defining a requirement dependency relation,* $\mathcal{P} = \langle P_c, P_t, T_e, T_i, f, f_\circ, \rho, \mathcal{L}, m_0 \rangle$ *is obtained by the union of each of the singleton CoPNs,* $\mathcal{P} = \textbf{\textit{union}}(\{\mathcal{C}_A, \mathcal{C}_B\})$*, and the application of the* $\textbf{\textit{ext}}_Q$ *and* $\textbf{\textit{cons}}_Q$ *functions to* $\mathcal{P}$*.*

The $\textbf{\textit{ext}}_Q$ *function is defined as* $\textbf{\textit{ext}}_Q(\mathcal{P}, \langle Q, \mathcal{C}_A, \mathcal{C}_B \rangle) = \mathcal{P}'$*, between two singleton CoPNs* $\mathcal{C}_A = \langle P_{c_A}, P_{t_A}, T_{e_A}, T_{i_A}, f_A, f_{\circ_A}, \rho_A, \mathcal{L}_A, m_{0_A}, \Sigma_A, \lambda_A \rangle$ *and* $\mathcal{C}_B = \langle P_{c_B}, P_{t_B}, T_{e_B}, T_{i_B}, f_B, f_{\circ_B}, \rho_B, \mathcal{L}_B, m_{0_B}, \Sigma_B, \lambda_B \rangle$*, where* $\mathcal{C}_A, \mathcal{C}_B \subset \mathcal{P} = \langle P_c, P_t, T_e, T_i, f, f_\circ, \rho, \mathcal{L}, m_0, \Sigma, \lambda \rangle$ *and* $\mathcal{P}' = \langle P_c, P_t, T_e, T_i', f', f_\circ', \rho', \mathcal{L}, m_0, \Sigma, \lambda' \rangle$*, such that:*

$$T_i' = T_i \cup \{t'\}$$

$$\lambda'(e) = \begin{cases} \lambda(e) & \text{if } e \in P_c \cup P_t \cup T_e \cup T_i \\ deac(\mathtt{A}) & \text{if } e = t' \end{cases}$$

$$\rho'(t) = \begin{cases} \rho(t) & \text{if } t \in T_e \cup T_i \\ 2 & \text{if } t = t' \end{cases}$$

$$f'(t,p,l) = \begin{cases} f(t,p,l) & \text{if } t \in T_e \cup T_i \wedge p \in P_c \cup P_t \wedge l \in \mathcal{L} \\ 1 & \text{if } t = t' \wedge \lambda(p) = Pr(\neg \mathtt{A}) \wedge l \in \mathcal{L} \\ 1 & \text{if } t = t' \wedge \lambda(p) = \mathtt{A} \wedge l \in \mathcal{L} \\ 0 & \text{otherwise} \end{cases}$$

$$f'(p,t,l) = \begin{cases} f(p,t,l) & \text{if } p \in P_c \cup P_t \wedge t \in T_e \cup T_i \wedge l \in \mathcal{L} \\ 1 & \text{if } \lambda(p) = \mathtt{A} \wedge t = t' \wedge l \in \mathcal{L} \\ 0 & \text{otherwise} \end{cases}$$

$$f'_{\circ}(p,t) = \begin{cases} f_{\circ}(p,t) & \text{if } p \in P_c \cup P_t \ \wedge\ t \in T_e \cup T_i \\ 1 & \text{if } \lambda(p) = \mathtt{B} \wedge t = t' \\ 1 & \text{if } \lambda(p) = Pr(\neg \mathtt{A}) \wedge t = t' \\ 0 & \text{otherwise} \end{cases}$$

The $\mathtt{ext}_Q$ *function introduces a deactivation transition,* t'*, for the target context. This transition requests the deactivation of the target* A *whenever the source context* B *is inactive and* A *remains active.*

The $\mathtt{cons}_Q$ *function is defined as* $\mathtt{cons}_Q(\mathcal{P}, \langle Q, \mathcal{C}_A, \mathcal{C}_B \rangle) = \mathcal{P}'$*, between two singleton CoPNs* $\mathcal{C}_A = \langle P_{c_A}, P_{t_A}, T_{e_A}, T_{i_A}, f_A, f_{\circ_A}, \rho_A, \mathcal{L}_A, m_{0_A}, \Sigma_A, \lambda_A \rangle$ *and* $\mathcal{C}_B = \langle P_{c_B}, P_{t_B}, T_{e_B}, T_{i_B}, f_B, f_{\circ_B}, \rho_B, \mathcal{L}_B, m_{0_B}, \Sigma_B, \lambda_B \rangle$*, where* $\mathcal{C}_A, \mathcal{C}_B \subset \mathcal{P}$ $\langle P_c, P_t, T_e, T_i, f, f_{\circ}, \rho, \mathcal{L}, m_0, \Sigma, \lambda \rangle$ *and* $\mathcal{P}' = \langle P_c, P_t, T_e, T_i, f', f_{\circ}, \rho, \mathcal{L}, m_0, \Sigma, \lambda \rangle$*, such that*

$$f'(p,t,l) = \begin{cases} 1 & \text{if } \mathtt{A} \in t\bullet \wedge \mathtt{A} \notin \bullet t \ \wedge\ \lambda(p) = \mathtt{B} \ \wedge\ l \in \mathcal{L} \quad (6.13) \\ f(p,t,l) & \text{otherwise} \quad (6.14) \end{cases}$$

$$f'(t,p,l) = \begin{cases} 1 & \text{if } \mathtt{A} \in t\bullet \wedge \mathtt{A} \notin \bullet t \ \wedge\ \lambda(p) = \mathtt{B} \ \wedge\ l \in \mathcal{L} \quad (6.15) \\ f(t,p,l) & \text{otherwise} \quad (6.16) \end{cases}$$

The arcs introduced by $\mathtt{cons}_Q$ *represent that, for every transition activating* A*, the transition is enabled if and only if* B *is active.*

Figure 6.6 illustrates a CoPN representing the requirement dependency relation $\langle Q, \mathcal{C}_{\mathtt{N}}, \mathcal{C}_{\mathtt{C}} \rangle$ between the NLBS (N) and CONNECTIVITY (C) contexts of the maps application (Section 2.3.3).

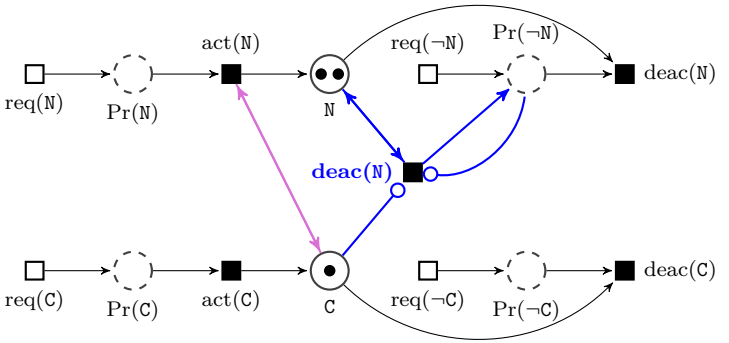


Figure 6.6: Requirement dependency relation (N–◀C).

Requirement dependency relations are commonly used when one situation can occur only if another is already taking place. In the maps application, the position calculation of the NLBS service is based on the location inferred from a local network connection. Hence, NLBS requires CONNECTIVITY, as shown

in Figure 6.6. If there is no connectivity available, then it is not possible to retrieve the position from the local network and another positioning service must be used.

Conjunction dependency relation

A *conjunction* dependency relation is somewhat different from the previously presented dependency relations. A conjunction dependency relation is defined for a given finite set of contexts, making behavioral adaptations associated to all of the contexts in the set become available only when all contexts in the set are active, otherwise the adaptations are not available. The conjunction dependency relation has a similar semantics as the logic *and* ($\wedge$) operator.

The conjunction dependency relation gathers the interaction between a set of contexts. To mange such interaction we introduce a new context place, labeled $\mathtt{A}_1 \cdots \mathtt{A}_n$, representing the contexts involved in the dependency relation as a unit. The context place $\mathtt{A}_1 \cdots \mathtt{A}_n$ cannot be directly manipulated —that is, it cannot be activated by firing its external transitions. Instead the context place is automatically activated whenever all contexts $\mathtt{A}_j$ become active. As a consequence if either of the source contexts $\mathtt{A}_j$ involved in the dependency relation are deactivated, it must be ensured that the context place $\mathtt{A}_1 \cdots \mathtt{A}_n$ also becomes inactive.

Definition 6.17 (Conjunction). *The conjunction dependency relation for a set of contexts ($\rightarrow (\mathtt{A}_1 \cdots \mathtt{A}_n)$) is defined as a tuple $\langle \wedge, \mathcal{C}_{A_1}, \ldots, \mathcal{C}_{A_n} \rangle$, where $\mathcal{C}_{A_j}$ are pairwise different singleton CoPNs, for $1 \leq j \leq n$. The composed CoPN defining the conjunction dependency relation, $\mathcal{P} = \langle P_c, P_t, T_e, T_i, f, f_\circ, \rho, \mathcal{L}, m_0, \Sigma, \lambda \rangle$, is obtained by the union of all singleton CoPNs, $\mathcal{P} = \boldsymbol{union}(\{\mathcal{C}_{A_1}, \ldots, \mathcal{C}_{A_n}\})$, followed by the application of the $\boldsymbol{ext}_\wedge$ and $\boldsymbol{cons}_\wedge$ functions to $\mathcal{P}$.*

The $\boldsymbol{ext}_\wedge$ function is defined as $\boldsymbol{ext}_\wedge(\mathcal{P}, \langle \wedge, \mathcal{C}_{A_i}, \ldots, \mathcal{C}_{A_n} \rangle) = \mathcal{P}'$, between the singleton CoPNs $\mathcal{C}_{A_j} = \langle P_{c_{A_j}}, P_{t_{A_j}}, T_{e_{A_j}}, T_{i_{A_j}}, f_{A_j}, f_{\circ_{A_j}}, \rho_{A_j}, \mathcal{L}_{A_j}, m_{0_{A_j}}, \Sigma_{A_j}, \lambda_{A_j} \rangle$ such that for $1 \leq j \leq n$, $\mathcal{C}_{A_j} \subset \mathcal{P} = \langle P_c, P_t, T_e, T_i, f, f_\circ, \rho, \mathcal{L}, m_0, \Sigma, \lambda \rangle$ and $\mathcal{P}' = \langle P'_c, P'_t, T_e, T'_i, f', f'_\circ, \rho', \mathcal{L}, m_0, \Sigma', \lambda' \rangle$, where:

$$\Sigma' = \Sigma \cup \{\mathtt{A}_1 \cdots \mathtt{A}_n, Pr(\neg \mathtt{A}_1 \cdots \mathtt{A}_n), act(\mathtt{A}_1 \cdots \mathtt{A}_n), deac(\mathtt{A}_1 \cdots \mathtt{A}_n)\}$$

$$P'_c = P_c \cup \{p'\}$$

$$P'_t = P_t \cup \{p''\}$$

$$T'_i = T_i \cup \{t', t'', t'''\}$$

$$\lambda'(e) = \begin{cases} \lambda(e) & \text{if } e \in P_c \cup P_t \cup T_e \cup T_i \\ deac(\mathtt{A}_1 \cdots \mathtt{A}_n) & \text{if } e = t'' \vee e = t''' \\ act(A_1 \cdots A_n) & \text{if } e = t' \\ A_1 \cdots A_n & \text{if } e = p' \\ Pr(\neg A_1 \cdots A_n) & \text{if } e = p'' \end{cases}$$

$$\rho'(t) = \begin{cases} \rho(t) & \text{if } t \in T_e \cup T_i \\ 2 & \text{if } t = t' \vee t = t'' \vee t = t''' \end{cases}$$

$$f'(p,t,l) = \begin{cases} f(p,t,l) & \text{if } p \in P_c \cup P_t \wedge t \in T_e \cup T_i \wedge l \in \mathcal{L} \\ 1 & \text{if } \lambda(p) = A_j \text{ for } 1 \leq j \leq n \wedge t = t' \wedge l \in \mathcal{L} \\ 1 & \text{if } p = p'' \wedge t = t'' \wedge l \in \mathcal{L} \\ 1 & \text{if } p = p' \wedge t = t'' \wedge l \in \mathcal{L} \\ 1 & \text{if } p = p'' \wedge t = t''' \wedge l \in \mathcal{L} \\ 0 & \text{otherwise} \end{cases}$$

$$f'(t,p,l) = \begin{cases} f(t,p,l) & \text{if } p \in P_c \cup P_t \wedge t \in T_e \cup T_i \wedge l \in \mathcal{L} \\ 1 & \text{if } t = t' \wedge \lambda(p) = A_j \text{ for } 1 \leq j \leq n \wedge l \in \mathcal{L} \\ 1 & \text{if } t = t' \wedge p = p' \wedge l \in \mathcal{L} \\ 0 & \text{otherwise} \end{cases}$$

$$f'_\circ(p,t) = \begin{cases} f_\circ(p,t) & \text{if } p \in P_c \cup P_t \wedge t \in T_e \cup T_i \\ 1 & \text{if } p = p' \wedge t = t' \\ 1 & \text{if } p = p' \wedge t = t''' \\ 0 & \text{otherwise} \end{cases}$$

The $\mathtt{ext}_\wedge$ *function introduces a new context place,* p' *(labeled* $\mathtt{A}_1 \cdots \mathtt{A}_n$*) to represent that all contexts involved in the dependency relation are currently active. The introduced internal transitions and temporary place are used to manage the activation and deactivation of the new context place. The activation transition is enabled whenever all of the contexts composing the conjunction are marked, and the new context place is not (this is done to avoid an infinite sequence of firings of thet' transition).*

The $\mathtt{cons}_\wedge$ *function is defined as* $\mathtt{cons}_\wedge(\mathcal{P}, \langle \wedge, \mathcal{C}_{A_1}, \ldots, \mathcal{C}_{A_n} \rangle) = \mathcal{P}'$*, for the singleton CoPNs* $\mathcal{C}_{A_j} = \langle P_{c_{A_j}}, P_{t_{A_j}}, T_{e_{A_j}}, T_{i_{A_j}}, f_{A_j}, f_{\circ_{A_j}}, \rho_{A_j}, \mathcal{L}_{A_j}, m_{0_{A_j}}, \Sigma_{A_j}, \lambda_{A_j} \rangle$*, where for* $1 \leq j \leq n$*,* $\mathcal{C}_{A_j} \subset \mathcal{P} = \langle P_c, P_t, T_e, T_i, f, f_\circ, \rho, \mathcal{L}, m_0, \Sigma, \lambda \rangle$ *and* $\mathcal{P}' = \langle P_c, P_t, T_e, T_i, f', f_\circ, \rho, \mathcal{L}, m_0, \Sigma, \lambda \rangle$*, such that:*
$\forall\, p \in P_c \cup P_t, \forall\, t \in T_e \cup T_i$ *and* $\forall l \in \mathcal{L}$*,*

$$f'(p,t,l) = f(p,t,l)$$

$$f'(t,p,l) = \begin{cases} 1 & \text{if } A_j \in t\bullet \wedge A_j \notin \bullet t \text{ for } 1 \leq j \leq n \\ & \wedge\, p = p'' \wedge\, l \in \mathcal{L} \quad (6.17) \\ f(t,p,l) & \text{otherwise} \quad (6.18) \end{cases}$$

The $\texttt{cons}_\wedge$ *function adds an arc from the deactivation of each of the component contexts involved in the of the conjunction dependency relation to prepare to deactivate the temporary place of the context place representing the conjunction of all contexts. If such a context place is marked the enabled deactivation the new introduced transition* t'' *by the* $\texttt{ext}_\wedge$ *function, the right-most transition labeled* $deac(A_1 \cdots A_n)$*, in Figure 6.7. Transition* t''' *introduced by the* $\texttt{ext}_\wedge$ *function, the bottom-most transitions labeled* $deac(A_1 \cdots A_n)$ *in Figure 6.7, is enabled if the context place is not marked. These interaction states that if one of the component contexts becomes inactive, the conjunction is also made inactive.*

Figure 6.7 illustrates the CoPN representing the conjunction dependency relation $\langle \wedge, \mathcal{C}_\texttt{F}, \mathcal{C}_\texttt{C} \rangle$ between the `FRIENDS (F)` and `CONNECTIVITY (C)` contexts of the maps application (Section 2.3.3).

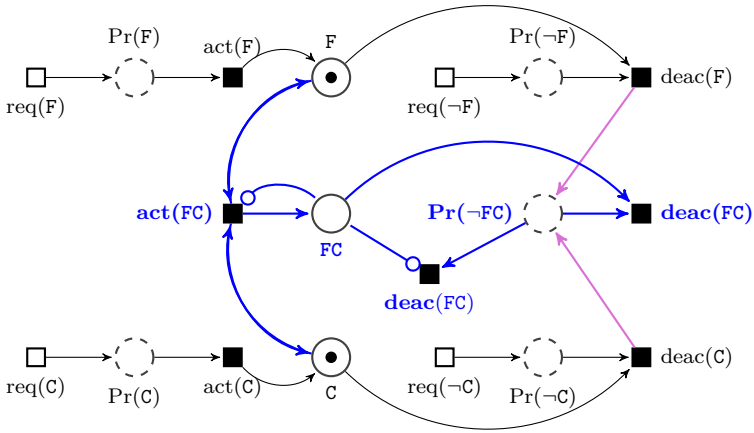


Figure 6.7: Conjunction dependency relation ($\rightarrow$(F C)).

Composing general CoPNs

The context dependency relations defined in Definitions 6.13 through 6.17 describe how to generate a composed CoPN from several existing singleton CoPNs. The composition operator $\circ$ of CoPNs given in Definition 6.6 describes how more general CoPNs can be obtained by the application of the `ext` (Definition 6.11) and `cons` (Definition 6.12) functions over any finite set of singleton CoPN and any finite set of context dependency relations defined over such contexts. Theorem 6.1 demonstrates that the obtained CoPN from the composition of a set of

singleton CoPNs and context dependency relations is always the same regardless of the order in which the ext, cons functions are applied for each of the context dependency relations.

Theorem 6.1. *Given $\mathcal{S} \subset \mathscr{S}$ a set of singleton CoPNs and $\mathcal{R} \subset \mathscr{R}$ a set of dependency relations between the CoPNs in $\mathcal{S}$. The CoPN $\mathcal{P} = \circ(\mathcal{S}, \mathcal{R})$ is always the same regardless of the order in which the functions $\mathtt{ext}_R$ and $\mathtt{cons}_R$ are applied for the context dependency relations $\langle R, \mathcal{C}_1, \ldots, \mathcal{C}_n \rangle$ in $\mathcal{R}$.*

Proof. Let us suppose a context dependency relation $\langle R, \mathcal{C}_{\mathtt{A}_1}, \ldots, \mathcal{C}_{\mathtt{A}_n}, \mathcal{C}_{\mathtt{B}} \rangle$ with associated functions $\mathtt{ext}_R$ and $\mathtt{cons}_R$, where the singleton CoPNs $\mathcal{C}_{\mathtt{A}_j}$ are the source contexts of the dependency relation for $1 \leq j \leq n$, and $\mathcal{C}_{\mathtt{B}}$ is the target singleton CoPN of the dependency relation.

1. The $\mathtt{ext}_R$ functions add transitions and places specific to each context dependency relation R. Note from Definitions 6.13 through 6.17, that there are no restrictions imposed on the addition of such elements. This implies that all elements specified for the context dependency relations $R \in \mathcal{R}$ are always added regardless of the order in which the functions $\mathtt{ext}_R$ are applied.

2. The $\mathtt{cons}_R$ functions add arcs between places and transitions of a CoPN $\mathcal{P}$. However, as the addition of such arcs is restricted by the characteristics of the input and output sets of transitions, we prove that for every context dependency relation, if for a transition t the function $\mathtt{cons}_R$ introduces an arc, then there is is no other context dependency relation that adds arcs to t which would cause $\mathtt{cons}_R$ not to add this arc any longer. From Definitions 6.13 through 6.17 it is possible to differentiate two kinds of enabling conditions over the transitions.

   $\underline{\mathtt{A}_j \in t\bullet \wedge \mathtt{A}_j \notin \bullet t}$: This enabling condition is applicable for the transitions t for which a source context place $\mathtt{A}_j$ is an output and is not an input. This is the enabling condition of Equations (6.1), (6.2), (6.8), (6.11), (6.13), (6.13), and (6.17).

   Equations (6.1) and (6.2), used in $\mathtt{cons}_E$, only add an inhibitor arc from context places to transitions. Adding such arcs does not validates or invalidates the enabling condition.

   Equations (6.8) and (6.7) for the $\mathtt{cons}_C$ function, (6.10) and (6.11) for the $\mathtt{cons}_I$, and (6.17) for the $\mathtt{cons}_\wedge$ respectively add arcs to the preparing to activate or preparing to deactivate temporary places of the target context. Thus they do not invalidate enabling condition.

   Equations (6.6) from the $\mathtt{cons}_C$ and (6.15) from $\mathtt{cons}_R$ define the target context place B as an output of the transition. Application of this function could enable this condition ($\mathtt{A}_j \in t\bullet \wedge \mathtt{A}_j \notin \bullet t$) on the transition for subsequent applications of the $\mathtt{cons}_R$ functions. However we recognize this is not possible because every time Equations (6.6)

and (6.15) are applied, so are Equations (6.4) and (6.13), which also define the target context place B as an input of the transition, thus invalidating the enabling condition. If there is a transition for which this enabling condition is valid, application of Equations (6.4), (6.13), (6.6) or (6.15) does not invalidate it. This is due to the arcs are being added over the target context, while the restriction is valid over output transition of the source context places.

The application of the $\mathtt{cons}_E$, $\mathtt{cons}_C$, $\mathtt{cons}_I$, $\mathtt{cons}_Q$, and $\mathtt{cons}_\wedge$ functions does not invalidate the enabling condition $\mathtt{A}_j \in t\bullet \wedge \mathtt{A}_j \notin \bullet t$.

$\mathtt{A}_j \in \bullet t \wedge \mathtt{A}_j \notin t\bullet \wedge \mathtt{B} \notin \circ t$: This enabling function is applicable for the transitions t for which a source context place $\mathtt{A}_j$ is an input and not an output, and the target context place B is not an inhibitor input. This is the enabling condition of Equations (6.4), (6.6), (6.7), and (6.11).

Equations (6.1) and (6.2), used in $\mathtt{cons}_E$, add an inhibitor arc from context places to transitions for which the source context is an output. Hence, adding such arcs does not invalidates this enabling condition.

None of the Equations (6.4) through (6.17) are used in functions $\mathtt{cons}_C$, $\mathtt{cons}_I$, $\mathtt{cons}_Q$, or $\mathtt{cons}_\wedge$ to introduce regular arcs to the CoPN, thus they do not invalidate the enabling condition of the target context not being an inhibitor input of a transition. Therefore, similar to the previous case, only Equations (6.4), (6.13), (6.6) or (6.15) from $\mathtt{cons}_C$ and $\mathtt{cons}_Q$ add arcs to the CoPN such that context places become inputs of a transition or outputs of a transition. We can then use the same argument used for the previous enabling condition to guarantee that these equations do not invalidate $\mathtt{A}_j \notin t\bullet$, nor do they validate the $\mathtt{A}_j \in \bullet t$ enabling condition.

The application of the $\mathtt{cons}_E$, $\mathtt{cons}_C$, $\mathtt{cons}_I$, $\mathtt{cons}_Q$, and $\mathtt{cons}_\wedge$ functions does not invalidate the enabling condition $\mathtt{A}_j \in \bullet t \wedge \mathtt{A}_j \notin t\bullet \wedge \mathtt{B} \notin \circ t$

Then the $\mathtt{ext}_E$, $\mathtt{ext}_C$, $\mathtt{ext}_I$, $\mathtt{ext}_Q$, and $\mathtt{ext}_\wedge$ and the $\mathtt{cons}_E$, $\mathtt{cons}_C$, $\mathtt{cons}_I$, $\mathtt{cons}_Q$, and $\mathtt{cons}_\wedge$ functions can be applied in any order without changing the outcome of the ext and cons functions. □

Examples 6.2 and 6.3 illustrate the composition process CoPN for different sets of context dependency relations.

Example 6.2. Let us consider three contexts A, B, and C and two context dependency relations $R_1 = \langle I, \mathcal{C}_\mathtt{A}, \mathcal{C}_\mathtt{B}\rangle$ and $R_2 = \langle C, \mathcal{C}_\mathtt{A}, \mathcal{C}_\mathtt{C}\rangle$ with corresponding CoPNs $\mathcal{P}_1 = \circ(\{\mathcal{C}_\mathtt{A}, \mathcal{C}_\mathtt{C}\}, \{R_1\})$ and $\mathcal{P}_2 = \circ(\{\mathcal{C}_\mathtt{A}, \mathcal{C}_\mathtt{C}\}, \{R_2\})$. Suppose that for the implication dependency relation context A is activated three times (and hence context B), and that for the causality dependency relation context A is activated one time (and hence context C). Let us take the composition of these two context dependency relations into a single CoPN, $\mathcal{P} = \circ(\{\mathcal{C}_\mathtt{A}, \mathcal{C}_\mathtt{B}, \mathcal{C}_\mathtt{A}, \mathcal{C}_\mathtt{C}\}, \{R_1, R_2\})$. Note that in this case, the transition introduced by the $\mathtt{ext}_C$ function (i.e., the

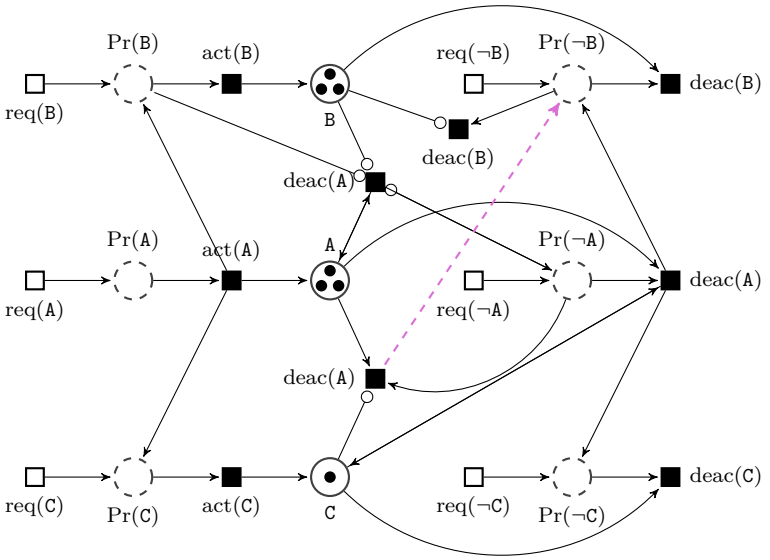


Figure 6.8: Multiple activations unification.

bottom-most deac(A) transition) is a transition deactivating the source context of the implication dependency relation, and hence it must request the deactivation of the target context B. This is guaranteed by the $\texttt{cons}_I$ function with the introduction of the (deac(A), Pr(¬B)), shown with a dashed line in Figure 6.8.[2]

Figure 6.8 shows the composed CoPN $\mathcal{P}$. Note that context place A is marked with 3 tokens in $\mathcal{P}$ according to the definition of the initial marking given in Definition 6.10, where $m_0(\texttt{A}, black)$=max$\{m_{0_1}(\texttt{A}, black), m_{0_2}(\texttt{A}, black)\}$. This marking of A allows to successfully respond to all request for deactivations of A as for each of the independent CoPNs.

If we request the deactivation of A, a token will be added to place Pr(¬A) enabling the right-most deac(A) transition. Firing of this transition adds tokens to Pr(¬ B) and Pr(¬C), which respectively enables the deac(B) and deac(C) transitions. Firing of each of these transitions removes a token for each of the context places B and C reaching a marking m, where $m(\texttt{A}, black) = 2$, $m(\texttt{B}, black) = 2$, and $m(\texttt{C}, black) = 0$. An additional request to deactivate A, as before, adds a token to place Pr(¬A), enabling the bottom most deac(A) transition (the right-most deac(A) transition is not enabled because C is not active anymore). Firing of this transition adds a token place Pr(¬ B) enabling transition deac(B). Firing this transition leads to a marking m_2 where $m_2(\texttt{A}, black) = 1$ and $m_2(\texttt{A}, black) = 1$. An extra deactivation of A follows the same process, leading to an empty marking.

[2]Here the arc is dashed as a means to identify it easily, this convention has no semantics in CoPN.

As this example shows, taking the maximum of the markings for places when taking the union of singleton CoPNs preserves the intended interaction between the contexts.

This example also illustrates the importance of defining the composition operator as a composition of the `union`, `ext`, and `cons` functions. Firstly, the `union` function fuses all singleton CoPNs that dependency relations may have in common, and thus unifies all singleton CoPNs in a single CoPN definition. Secondly, the `ext` function ensures that all transitions and places are added for the specific cases of interaction specified by context dependency relations. Thirdly, the `cons` function adds required arcs for the general interactions specified by context dependency relations.

Note that if we would add required places, arcs, and transition for each context dependency relation in one single function, the CoPN in Figure 6.8 would not have the dashed arc if the constraining function of the implication dependency relation would be applied prior to that of the causality dependency relation, which would cause an inconsistency in the system. Indeed, in such a case, it would be possible to reach a situation in which the source context of the implication dependency relation, `A`, is not active, but the target context `B` is, for example, by activating `A` twice and deactivating `C` once, and deactivating `A` twice. The second deactivation of `A` would not request the deactivation of `B`, even when the activation of the later context was due to a situation in which both `A` and `B` should be active. Behavioral inconsistencies could arise if the behavior introduced by context `B` reuses that introduced by context `A`.

Example 6.3. As an illustration of composing general CoPNs, let us take two CoPNs. A first CoPN $\mathcal{P}_1$ composed of two contexts `NLBS (N)` and `Positioning (Pos)`, with respective singleton CoPNs $\mathcal{C}_{\texttt{N}}$ and $\mathcal{C}_{\texttt{Pos}}$, and an implication dependency relation $\langle I, \mathcal{C}_{\texttt{N}}, \mathcal{C}_{\texttt{Pos}} \rangle$, and a second CoPN $\mathcal{P}_2$ composed of two contexts `NLBS (N)` and `Connectivity (C)`, with respective singleton CoPNs $\mathcal{C}_{\texttt{N}}$ and $\mathcal{C}_{\texttt{C}}$, and a requirement dependency relation $\langle Q, \mathcal{C}_{\texttt{N}}, \mathcal{C}_{\texttt{C}} \rangle$. For the purpose of these example, let us assume that the two CoPNs $\mathcal{P}_1$ and $\mathcal{P}_2$ have an empty initial marking. These two CoPNs can be composed into a single CoPN, $\mathcal{P}$, following the definition of the $\circ$ composition operator, the composed CoPN is defined by $\mathcal{P} = \circ(\{\mathcal{C}_{\texttt{N}}, \mathcal{C}_{\texttt{Pos}}, \mathcal{C}_{\texttt{C}}\}, \{\langle I, \mathcal{C}_{\texttt{N}}, \mathcal{C}_{\texttt{Pos}} \rangle, \langle Q, \mathcal{C}_{\texttt{N}}, \mathcal{C}_{\texttt{C}} \rangle\})$ is obtained by the composition of the two existing CoPNs. The visual representation of the resulting CoPN is shown in Figure 6.9.

As there is a singleton context that is shared by both context dependency relations, the first step in the composition is to fuse the two corresponding singleton CoPNs into a single context. Second, we add all required transitions and arcs by both context dependency relations using the functions $\texttt{ext}_Q$ and $\texttt{ext}_I$. Each of these functions add a transition labeled deac(`N`), the top-most transition for $\texttt{ext}_Q$ and the bottom-most transition for $\texttt{ext}_I$. Finally, we add additional arcs by the application of the $\texttt{cons}_I$ and $\texttt{cons}_Q$ functions. The former function adds the two arcs (act(`N`), Pr(`Pos`)) and (deac(`N`), Pr($\neg$`Pos`)) according to Equations (6.10) and (6.11), while the later function adds the arcs

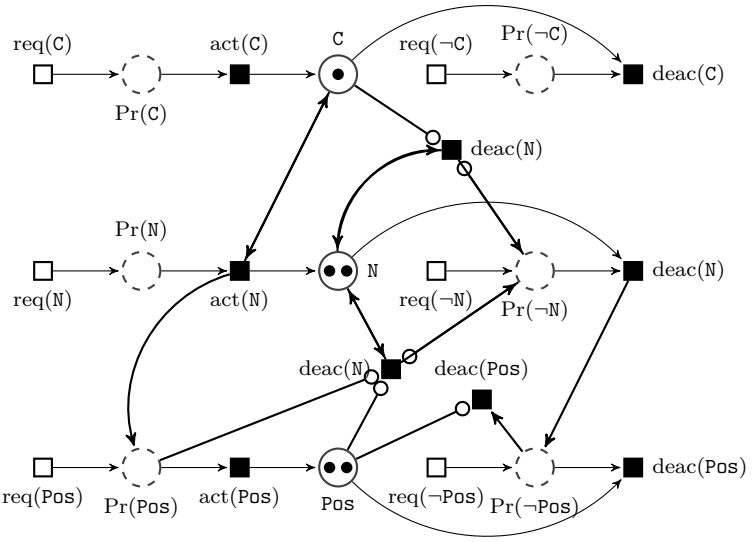


Figure 6.9: Composition of two CoPNs $\mathcal{P}_1 = \circ(\{\mathcal{C}_{\mathtt{N}}, \mathcal{C}_{\mathtt{Pos}}\}, \{\langle I, \mathcal{C}_{\mathtt{N}}, \mathcal{C}_{\mathtt{Pos}}\rangle\})$ and $\mathcal{P}_2 = \circ(\{\mathcal{C}_{\mathtt{N}}, \mathcal{C}_{\mathtt{C}}\}, \{\langle Q, \mathcal{C}_{\mathtt{N}}, \mathcal{C}_{\mathtt{C}}\rangle\})$ with an implication ($\mathtt{N}$–▶$\mathtt{Pos}$) and a requirement ($\mathtt{N}$–◀$\mathtt{C}$).

(act(N), C) and (C, act(N)) according to Equations 6.13 and 6.15. Note that in this particular case there are no introduced transitions by the $\mathtt{ext}_I$ or $\mathtt{ext}_Q$ such that new arcs need to be added than those already existing for each of the individual CoPNs $\mathcal{P}_1$ and $\mathcal{P}_2$.

Correctness of the composition operator

Definitions 6.13 through 6.17 described how for each of the context dependency relations, the functions $\mathtt{ext}_R$ and $\mathtt{cons}_R$ introduce places, transitions, and arcs to a CoPN. Such new elements are added in order to provide the desired behavior of each context dependency relation.

The question that still remains is whether the $\mathtt{ext}_R$ and $\mathtt{cons}_R$ functions effectively exhibit the intended behavior of the context dependency relations $\langle R, \mathcal{C}_1, \ldots, \mathcal{C}_n\rangle$, for any given CoPN. Below, we provide an informal argumentation about the validity of this statement. Chapter 7 introduces analysis techniques for CoPNs, which allow a CoPN to be tested, for example to verify if it satisfies the expected behavior of the context dependency relations composing it.

To guarantee the behavior of context dependency relations for any CoPN we divided their composition in two. For any context dependency relation $\langle R, \mathcal{C}_1, \ldots, \mathcal{C}_n\rangle$, the $\mathtt{ext}_R$ functions add the necessary elements between the

singleton CoPNs involved in such a context dependency relation. The added elements take into account specific cases of the interaction between the input singleton CoPNs. The $\texttt{cons}_R$ functions add the necessary arcs to the complete CoPN, such that the general behavior of each context dependency relation is satisfied.

Exclusion dependency relations do not require any specific interaction between the contexts involved in the context dependency relation. The only imposed constraint is that the contexts involved in an exclusion dependency relation cannot be active simultaneously. The $\texttt{cons}_E$ function ensures this interaction by adding inhibitor arcs from the context place of *each* context to *all* activation transitions of context places of the other context. This property can be seen in Equation (6.1) (and equivalently in Equation (6.2)), where for *every* transition for which the source (resp. target) context place is an output place and not an input place an inhibitor arc is added from the target (resp. source) context place. This property is verified after all transitions have been added in the CoPN, thus we know that it takes into account all possible activations of each context, thus ensuring that the transition is enabled if and only if the other context is not active.

Causality dependency relations add a transition by means of the $\texttt{ext}_C$ function to deactivate the source context of the dependency relation if the target context is inactive. This can occur because the target context can be deactivated independently from the source context. The introduction of this transition is needed because it might be possible that the source context is active and the target context is not. The $\texttt{cons}_C$ function adds arcs to request the activation or deactivation of the target context every time the source context is activated or deactivated. This means that for *every* transition for which the source context place is an output and not an input (i.e., transitions adding tokens to the context place) an arc is added to the preparing to activate temporary place of the target context (Equation (6.8)). Similarly, for *every* transition of the CoPN for which the source context place is an input and not an output (i.e., transitions removing tokens from the context place) an arc is added to the preparing to deactivate temporary place of the target context (Equation (6.7)). Additionally, to ensure that the target context it is actually active and the request to deactivate it can be processed, all transitions of the CoPN deactivating the source context are enabled if and only if the target context place is marked (Equations (6.4) and (6.6)).

Implication dependency relations add a transition by means of the $\texttt{ext}_I$ function to deactivate the source context of the context dependency relation if the target context is not active nor preparing to activate. The target context of the implication dependency relation can be activated independently from the source context. As a result, it is not possible to assert that every deactivation of the target context should deactivate the source

context. Remember from Example 4.5 that such an assertion caused a cycle in the deactivation transitions of the source and target contexts involved in an implication dependency relation. However it is possible to assert that if the target context is inactive, the source context should also become inactive, as specified by the introduced deactivation transition. As it is the case for the causality dependency relation, the $\texttt{cons}_I$ function also adds arcs to request the activation and deactivation of the target context with *every* activation and deactivation of the source context. This is respectively managed by Equations (6.10) and (6.11).

Requirement dependency relations add a transition to automatically trigger the deactivation of the source context whenever the target context is inactive. This transition, introduced by the $\texttt{ext}_Q$ function, guarantees that the source context cannot be activated if the target context is inactive. The $\texttt{cons}_Q$ function ensures that *all* transitions for which the source context place is an output and not an input (i.e., transitions adding tokens to the source context place) are enabled if and only if the target context is active. This is ensured by adding the arcs described in Equations (6.13) and (6.15).

Conjunction dependency relations add the places and transitions by means of the $\texttt{ext}_\wedge$ function to manage the activation state of all the contexts involved in the dependency relation. The state of all contexts is represented by a new context place, labeled $A_1 \cdots A_n$. Tokens are added to this place only if all of the source context places are active. This situation is represented by a new activation transition in which each of the context places for the source contexts is an input and an output. The transition also has the new context place as an inhibitor input to ensure that the transition is only fired once. Deactivation of the new context place is done by requesting it through a newly introduced preparing to deactivate temporary place. Two transitions are added to handle the deactivations of the new context place. A first transition removes the tokens from the new context place whenever it is preparing to deactivate. If the new context place is not marked, then a second transition removes the token from the new preparing to deactivate temporary place. The $\texttt{cons}_\wedge$ function introduces arcs to ensure that for *every* transition deactivating one of the source contexts involved in the conjunction dependency relation, the request to deactivate the new context place is forwarded by adding a token to the new preparing to deactivate temporary place. These new arcs ensure that if one of the source contexts becomes inactive, then the new context place is also made inactive.

6.3 Managing Dynamic Behavioral Adaptations

CoPNs are not static structures. On the contrary, CoPNs make it possible to represent and track the changes that occur in the system's surrounding execution

environment. CoPNs can thus be used as run-time representation of contexts and their dynamic changes. The following description summarize how the state of contexts is encoded in a CoPN, and how it evolves according to its firing semantics (Section 6.1).

- In CoPNs changes in the surrounding execution environment of the system trigger changes in the activation state of its contexts. Such changes are associated with external transitions to activate or deactivate a context, depending on what is required by the sensed situation.
- External transition firings (may) enable internal transitions, which are processed using the semantics of reactive Petri nets and transition priorities. Eventually a marking is reached where no internal transitions can fire. Infinite sequences of internal transition firings cause inconsistencies in the system behavior, and are hence not allowed in CoPNs. Chapter 7 discusses how to identify context configurations in which this is possible, as a means to remove them from the application.
- After there are no more internal transitions to be fired, if there are marked temporary places, it means that one or more of the constraints imposed by context dependency relations are not satisfied, and therefore that context activation or deactivation may raise inconsistencies in the system behavior. To avoid this, the CoPN consequently rolls back all state changes made to the contexts since the firing of the external transition.
- After there are no more internal transitions to be fired, if on the other hand, there are no marked temporary places, then the context activation or deactivation does not lead to inconsistencies, and can be processed successfully.
- Other requests to activate and deactivate contexts can then be processed in response to external transition firings.

The internal semantics of CoPNs are processed according to the semantics of Colored Petri nets, reactive Petri nets, inhibitor arcs and transition priorities as follows:

- External transitions are always enabled since they are source transitions (i.e., they do not have any input places).
- An internal transition t is *enabled* for a color l if its input places p_{in} from regular arcs contain at least $f(p_{in}, t, l)$ tokens, its input places $p_\circ$ from inhibitor arcs are empty, and no other transition t' with a higher priority, $\rho(t') > \rho(t)$, is enabled.
- Internal transitions are fired with a *must* fire semantics. That is, if an internal transition is enabled it must fire. Whenever two internal transitions (with the same priority) are enabled simultaneously they fire non-deterministically.

- *Firing* of a transition t for a given color l modifies the state of the CoPN by removing $f(p_{in}, t, l)$ tokens from its input places p_{in}, and adding $f(t, p_{out}, l)$ tokens to its output places p_{out}.

6.3.1 Context Activation Semantics

In this section we formalize the activation semantics of CoPNs. As we will see, this semantics respects the intended interactions between contexts as defined by context dependency relations, and automatically manages inconsistencies that may arise through context activation or deactivation.

Every external transition firing may be followed by a sequence of internal transition firings, called a step. The reactive semantics of CoPNs force internal transitions to fire whenever they become active.

Definition 6.18. *Given two reachable marking multisets m, m' of a CoPN $\mathcal{P} = \langle P_c, P_t, T_e, T_i, f, f_\circ, \rho, \mathcal{L}, m_0, \Sigma, \lambda\rangle$. If $t_0 \in T_e$ and for $j = 1, \ldots, n$, $t_j \in T_i$, are such that $m[t_0\rangle m_1[t_1\rangle \ldots m_n[t_n\rangle m'$, the sequence of transitions $\Upsilon = t_0 t_1 \ldots t_n$, is called an **step** between m and m'.*

For two marking multisets m and m', if $m[\Upsilon\rangle m'$ we say that m' is reachable from m via the step Υ.

Definition 6.19. *Given a CoPN $\mathcal{P} = \langle P_c, P_t, T_e, T_i, f, f_\circ, \rho, \mathcal{L}, m_0, \Sigma, \lambda\rangle$, and m a reachable marking multiset of $\mathcal{P}$. If there exist an infinite sequence of internal transitions $t_1 t_2 \ldots$, such that for an external transition firing $t_0 \in T_e$, $m[t_0\rangle m'[t_1\rangle \ldots$. The sequence $\Upsilon = t_0 t_1 t_2 \ldots$, is called an **unstable** step.*[3]

In the remainder of this dissertation we will assume to have stable firing steps unless explicitly said otherwise.

Definition 6.20. *Given a CoPN $\mathcal{P} = \langle P_c, P_t, T_e, T_i, f, f_\circ, \rho, \mathcal{L}, m_0, \Sigma, \lambda\rangle$, a marking multiset $\tilde{m}$ of $\mathcal{P}$ is said to be **consistent** if and only if $\forall p \in P_t$ and $\forall\, l \in \mathcal{L}$, $m(p, l) = 0$.*

Definition 6.21. *Let $\tilde{m}$ be a consistent marking of a CoPN $\mathcal{P}$. A step Υ, from $\tilde{m}$ to a marking multiset m', $\tilde{m}[\Upsilon\rangle m'$, is called a* consistent step *if and only if m' is a consistent marking multiset of $\mathcal{P}$.*

The state of a CoPN can only be modified after an external request to activate or deactivate a context. Whenever one of these actions is triggered in the system for a particular context, `A`, the corresponding external transition (i.e., req(`A`) for activation, or req(¬`A`) for deactivation) is fired in the CoPN. The semantics of transitions priorities of CoPNs ensure that no other external transition is fired until the step has finished —that is, until there are no more internal transitions to be fired. The general process of context activation is intuitively described as follows:

[3] The concept of unstable step corresponds with that of divergent firing sequences of reactive Petri nets [61].

1. Before external transitions are fired, the set of currently active contexts (i.e., the consistent state of the CoPN) is saved as the *current marking* of the system.

2. Internal transitions are fired until no internal transition remains enabled. This process follows the reactive (Definition 5.16) and priority transitions (Definition 5.13) semantics of CoPNs. In addition, the state of every transition firing (marked contexts and enabled transitions) is saved as a means to backtrack in case of inconsistencies.

3. Once no internal transitions remain enabled, two possibilities of execution exist:
   a) If some temporary places remain marked this means there are inconsistencies, since not all initially enabled internal transitions have been fired. In that case the state of the CoPN is reverted to the first state of internal transition firings saved in step 2. Another sequence of internal transition firings is tried.
   b) If an inconsistency exists after firing all initially enabled internal transitions, *all* modifications made since the external transition firing are rolled back to the *current marking* saved in step 1. Another context activation or deactivation request corresponding to an external transition can be processed then. If after processing all possible sequences of the initially enabled internal transitions an inconsistency still exists, the reason why the activation or deactivation could not take place is saved.

4. If the system reaches a consistent state —no temporary places remain marked, then the saved *current marking* is updated with the marking of the CoPN. A trace of the fired internal transitions to reach the state is saved as a means of logging to track the interactions between contexts. Other context activation or deactivation requests corresponding to an external transition can then be processed.

Transition firing semantics in CoPNs are designed to ensure that given a particular state and action over a context, the result of executing such an action is always the same. This is corroborated later in this section by Theorem 6.4. We now formalize the firing semantics of CoPNs.

Definition 6.22. *The* state *of a COP system is defined as a 5-tuple* $\langle \mathcal{P}, \mathcal{E}, \mathcal{T}, \mathscr{T}, \tilde{m} \rangle$*, where* $\mathcal{P}$ *is the CoPN of the system,* $\mathcal{E} \subseteq T_i$ *is the set of enabled internal transitions,* $\mathcal{T} \subseteq T_i$ *is a stack of fired transitions of the system,* $\tilde{m}$ *is the consistent marking multiset of* $\mathcal{P}$*, and* $\mathscr{T} = \langle t, \tilde{\mathcal{E}}, m \rangle$ *is the initial state of the system before the internal transition* t *is fired, where* $\tilde{\mathcal{E}}$ *is the set of enabled transitions and* m *is the marking multiset of* $\mathcal{P}$ *before firing* t*.*

Notation 6.8. *To facilitate the definition of our semantics rules, we introduce the following notation.*

- $\texttt{marked}_m(P) = \{p \in P \,|\, m(p,l) > 0 \text{ for } l \in \mathcal{L}\}$ *is the subset of marked places of a set* $P \subseteq P_c \cup P_t$.

- $\texttt{enabled}_m(T) = \{t \in T \,|\, m[t\rangle\}$ *is the subset of enabled transitions of a set* $T \subseteq T_e \cup T_i$ *at a marking multiset* m.

- $\mathcal{T} \triangleleft t$ *appends transition* t *to the stack* $\mathcal{T}$.

- m *and* m' *represent marking multisets of the* CoPN $\mathcal{P}$.

- $\pi_i(\mathscr{T})$ *is the* i^{th} *projection of tuple* $\mathscr{T}$.

Notation 6.9.

- *The notation* $[m/m']\mathcal{P}$ *is used to represent a replacement of the marking* m *in the Petri net* $\mathcal{P}$ *for marking* m'.

External transition firing: occurs only at the beginning of a step Υ when $\mathcal{P}$ is in a consistent state $\tilde{m}$ (i.e., the marking of $\mathcal{P}$ is consistent) and $\mathcal{E}$ is empty (i.e., i.e. no internal transitions are enabled). After firing an external transition, the marking of the CoPN is modified, possibly enabling transitions in T_i. Such transitions become the elements in the priority set $\mathcal{E}$.

$$\frac{t \in T_e,\ \tilde{m}[t\rangle m'}{\langle \mathcal{P}, \phi, \phi, \phi, \tilde{m}\rangle \rightarrow \langle \mathcal{P}, \texttt{enabled}_{m'}(T_i), \phi, \phi, \tilde{m}\rangle} \tag{6.19}$$

Internal transition firing: If the set $\mathcal{E}$ is not empty, firing one of the internal transitions with highest priority yields a new marking m' of the CoPN $\mathcal{P}$. The state of the CoPN before firing transition t is saved (as a means to facilitate backtracking in case of inconsistencies). The new marking possibly enables some internal transitions in T_i, which become the elements of $\mathcal{E}$.

$$\frac{t \in \mathcal{E},\ t \notin \mathcal{T},\ m[t\rangle m'}{\langle \mathcal{P}, \mathcal{E}, \mathcal{T}, \mathscr{T}, \tilde{m}\rangle \rightarrow \langle \mathcal{P}, \texttt{enabled}_{m'}(T_i), \mathcal{T} \triangleleft t, \langle t, \mathcal{E}, m\rangle, \tilde{m}\rangle} \tag{6.20}$$

Evaluation termination: When there are no more internal transitions to be fired in the set $\mathcal{E}$, three cases are possible:

1. The first case occurs when for m' marking multiset of $\mathcal{P}$ there are marked temporary places and not all initially enabled internal transitions have been fired. In such a case all changes are rolled back to the state of the CoPN m saved before the firing of the first internal transition t:

$$\frac{\texttt{marked}_{m'}(P_t) \neq \phi,\ m = \pi_3(\mathscr{T}),\ \pi_2(\mathscr{T}) \nsubseteq \mathcal{T}}{\langle \mathcal{P}, \phi, \mathcal{T}, \mathscr{T}, \tilde{m}\rangle \rightarrow \langle [m'/m]\mathcal{P}, \phi, \mathcal{T}, \phi, \tilde{m}\rangle} \tag{6.21}$$

2. The second case occurs when for m' marking multiset of $\mathcal{P}$ there are temporary places marked and all transitions initially enabled have been fired, the CoPN is rolled back to its last consistent state $\tilde{m}$ and the firing request is signaled as denied:

$$\frac{\texttt{marked}_{m'}(P_t) \neq \phi,\ \pi_2(\mathscr{T}) \subseteq \mathcal{T}}{\langle \mathcal{P}, \phi, \mathcal{T}, \mathscr{T}, \tilde{m} \rangle \ \rightarrow\ \langle [m'/\tilde{m}]\mathcal{P}, \phi, \phi, \phi, \tilde{m} \rangle} \tag{6.22}$$

3. The third case occurs when for m' marking multiset of $\mathcal{P}$ no temporary places remain marked, we finish the step by turning the current marking m' of the CoPN into the new consistent state $\tilde{m}$:

$$\frac{\texttt{marked}_{m'}(P_t) = \phi}{\langle \mathcal{P}, \phi, \mathcal{T}, \mathscr{T}, \tilde{m} \rangle \ \rightarrow\ \langle \mathcal{P}, \phi, \phi, \phi, m' \rangle} \tag{6.23}$$

The following result demonstrates that given a consistent state of a CoPN $\mathcal{P}$, every step Υ enabled at such state is consistent.

Theorem 6.2. *Let $\mathcal{P}$ be a consistent CoPN. Every finite step Υ, triggered by a request to activate or deactivate a context $\mathtt{A} \subseteq \mathcal{P}$, is a consistent step.*

Proof. After firing the initial external transition labeled $req(\mathtt{A})$ or $req(\neg\mathtt{A})$, Reduction rule 6.19 marks one of the temporary places labeled $Pr(\mathtt{A})$ or $Pr(\neg\mathtt{A})$ respectively. When temporary places are marked we have one of two cases:

1) If no internal transition is enabled after the external transition firing (that is if $\mathcal{E} = \phi$), Reduction rule 6.22 is applied, which rolls back $\mathcal{P}$ to its original consistent state ($\tilde{m}$). By hypothesis, $\tilde{m}$ is a consistent state, thus Υ is a consistent step.

2) If on the contrary, there are internal transitions to be fired, one of them is fired by applying Reduction rule 6.20. Every time this rule is applied, the marking of $\mathcal{P}$ is updated to a new marking m', and the state of the system is saved. When eventually the set $\mathcal{E}$ becomes empty, one of the three Reduction rules 6.21 through 6.23 can by applied:

   case 1: Reduction rule 6.21 is applied when there are marked temporary places. However, not all internal transitions initially enabled have fired. In this case the state of the CoPN is rolled back to the initial state m (where the internal transitions were enabled). The initial fired transitions is saved in the set of fired transitions not to visit such a firing sequence again. Other transitions not in the set of fired transitions from the initial enabled internal transitions is fired.

   case 2: Reduction rule 6.22 is applied when there are marked temporary places and all internal transitions initially enabled have fired. The CoPN is then rolled back to its original consistent state, $\tilde{m}$, which by hypothesis is a consistent state. Thus, Υ is a consistent (empty) step.

   case 3: Reduction rule 6.23 is applied when no temporary places are marked. Then, the marking of the system is updated to the marking of the CoPN m'. Marking m' is consistent by Definition 6.20 (i.e., no temporary places are marked). Thus, Υ is a consistent step.

□

If a step Υ leads to an inconsistent state —it leads to reduction rule 6.22— then the system is oblivious to the step and the CoPN is rolled back to its initial state. Whenever a context activation or deactivation is disregarded because it leads to an inconsistent state, the reason why the context (de)activation did not take place is signaled. For example in the case of the requirement dependency relation between `NLBS (N)` and `CONNECTIVITY (C)`, if the latter context is not active, the outcome of a request to activate the former context would be "context `NLBS` cannot be activated because context `NLBS` is preparing to activate and cannot complete the operation (context `CONNECTIVITY` is inactive)".

Example 6.4. To demonstrate the dynamics of context activation and deactivation in CoPN, consider the sequence of commands $\sigma = \{$ `@activate(N)`, `@activate(C)`, `@activate(N)`, `@activate(N)`, `@deactivate(C)` $\}$ for the CoPN composed of an implication and a requirement context dependency relations shown in Figure 6.9.

Let us begin with an empty initial marking of the system, $m_0(p) = 0\ \forall p \in P$. The `@activate(N)` message triggers the firing of the external transition req(`N`), which when fired leads to a marking m such that $m(Pr(\texttt{N})) = 1$. In this marking none of the internal transitions is enabled. In particular act(`N`) is not enabled, since context `C` is not active. At this stage Reduction rule 6.22 is applicable and marking m is reverted to the last consistent marking of the system, the initial marking m_0.

From the initial marking, let us take a state of the system in which the context activations `@activate(C)`, `@activate(N)`, and `@activate(N)` have been processed completely. The CoPN reaches a consistent state m (illustrated in Figure 6.9), where $m(\texttt{C}) = 1$, $m(\texttt{N})=2$ and $m(\texttt{Pos})=2$. Execution of the `@deactivate(C)` command, triggers the firing of external transition $req(\neg\texttt{C})$. This yields a new marking m_1, where $m_1(Pr(\neg\texttt{C})) = 1$, $m_1(\texttt{C}) = 1$, $m_1(\texttt{Pos})=2$, and $m_1(\texttt{N}) = 2$. For this marking the only enabled internal transition is the deactivation transition $deac(\texttt{C})$. Firing this transition yields a marking m_2, where $m_2(\texttt{C}) = 0$, $m_2(\texttt{Pos}) = 2$, $m_2(\texttt{N}) = 2$, and $m_2(Pr(\neg\texttt{C})) = 0$. Here, transition $deac(\texttt{N})$ (between contexts `N` and `C`) becomes enabled because place `C` is no longer marked. Firing this transition yields a marking m_3, where $m_3(\texttt{N}) = 1$, $m_3(\texttt{Pos}) = 2$, and $m_3(Pr(\neg\texttt{Pos})) = 1$. At this point (the same) transition $deac(\texttt{N})$ is enabled, and $deac(\texttt{Pos})$ becomes enabled too. Since the two transitions have the same priority, they can fire in random order. Suppose that $deac(\texttt{Pos})$ fires first. This leads to a marking m_4, where $m_4(\texttt{N}) = 1$, and $m_4(\texttt{Pos}) = 1$. This marking does not enable any new transition, however, transition $deac(\texttt{N})$ is in the Σ priority set and must fire. The firing yields marking $m_5(\texttt{Pos}) = 1$, and $m_5(Pr(\neg\texttt{Pos})) = 1$, enabling transition $deac(\texttt{Pos})$. Firing the transition yields an empty marking, which means the Petri net has reached a consistent state again.

Theorem 6.3. *Let m be a consistent marking of a CoPN $\mathcal{P}$, and Υ a consistent step, $m[\Upsilon\rangle m'$. Then $\forall\, \Upsilon'$ firing step which is a re-ordering of the transition*

firings in Υ, $m[\Upsilon'\rangle m'$.

Proof. Let us take t, t' two transitions in Υ. Note that any permutation of Υ where t and t' are interchanged implies that the two transitions are enabled at the same time, otherwise, they can not be fired in different orders. Suppose by contradiction that there are two markings m and m' such that $m[\Upsilon\rangle m'$ where t fires before t', and $m[\Upsilon'\rangle m''$ where t' fires before t. Without loss of generality let us take $m[t_0\rangle \dots m_i[t\rangle \dots m_j[t'\rangle \dots m'$ and $m[t_0\rangle \dots m_i[t'\rangle \dots m_j[t\rangle \dots m''$. We can distinguish two cases.

case $\bullet t \cap \bullet t' \neq \phi$: Constraint RC_2 of reactive Petri nets tells us that $\bullet t = \bullet t'$ or there is no marking enabling both transitions. By hypothesis it is the cases that $m_i[t\rangle$ and $m_j[t'\rangle$, thus it must be that $\bullet t = \bullet t'$. If this is the case, t and t' execute the same action over a context and hence have the same label, so $t = t'$. No matter the order in which the transitions are fired, the result is always the same, thus $m' = m''$.

case $\bullet t \cap \bullet t' = \phi$: The two transitions are independent. Since one transition is enabled at marking m_i and the other subsequently at marking m_j, one firing does not interfere with the other, so the overall result of firing both transitions are be the same, thus $m' = m''$.

□

Theorem 6.2 provides guarantees that a CoPN remains consistent, regardless of which contexts are activated or deactivated in the system, and Theorem 6.3 demonstrates that for every two reachable marking multisets, re-orderings of the step leading from one marking to the other do not affect the result —that is, the same marking is always reached. Nonetheless, we still need to prove that regardless of the order in which internal transitions are fired, for a given step Υ and initial state m, the resulting marking m' is always the same. This is given by the following result.

Theorem 6.4. *Let* $\mathcal{P}$ *be a* CoPN *and* $\mathcal{E}$ *the non-empty set of internal transitions to be fired in* $\mathcal{P}$ *at a marking multiset* m*. Let* Υ *be a firing step taking all internal transitions in* $\mathcal{E}$*. Then* $\exists!\, m'$ *marking multiset, such that* $\forall\, \Upsilon'$ *firing step which is a permutation of the transitions in* Υ, $m[\Upsilon'\rangle m'$.

Proof. Let $\mathcal{E} = \{t_1, \dots, t_n\}$ the set of enabled transitions at a marking m. For every pair of transitions $t_q, t_j \in \mathcal{E}$ with $q, j \in \{1, \dots, n\}$, we differentiate three cases in which the firing of the transitions could take place:

1. $\bullet t_q \cap \bullet t_j = \phi$ (the transitions have no inputs in common): If it would be the case that $\bullet t_q \cap \bullet t_j \neq \phi$, the constraint rule RC_2 of reactive Petri nets states that either t_q and t_j have the same inputs, or there is no marking enabling both transitions. By hypothesis we know that $m[t_q\rangle$ and $m[t_j\rangle$, thus it must be that t_q and t_j have the same inputs. If this is the case, Definition 6.8 states that $t_q = t_j$, which is not possible because $\mathcal{E}$ is a set; thus $\bullet t_q \cap \bullet t_j = \phi$.

In this case $\forall\, t_q, t_j$ such that $\exists\, p \in t_q\bullet \cap \circ t_j$, then $\nexists\, m''$ marking such that $[t_q\rangle m''[t_j\rangle$. Firing transition t_q leads to Reduction rule 6.21. So transition t_j always fires before t_q. After firing t_j, t_q is enabled because they do not share any input places. Firing of t_q leads to marking m'. Since Reduction rule 6.21 ensures the firing order $\{t_j, t_q\}$ the only reachable marking is m'. If firing t_j disables t_q, then no matter the order in which the transitions fire, the firing is always rolled back to the last marking saved in the initial state of the system $\mathscr{T}$.

2. $\circ t_q \cap t_j\bullet = \phi \wedge t_q \bullet \cap \circ\, t_j = \phi$: Two situations are distinguished:

   (a) Suppose $\nexists\, t'$ such that $m[t_q t' t_j\rangle$ or $m[t_j t' t_q\rangle$
   Take $m[t\rangle m^2[t_j\rangle m^1$ and $m[t_j\rangle m^3[t_q\rangle m^4$.
   Suppose by contradiction that $m^1 \neq m^4$.
   $\forall\, p \in P,\ m^2(p) = m(p) - f(p, t_q) + f(t_q, p)$ and
   $m^1(p) = m^2(p) - f(p, t_j) + f(t_j, p)$.
   Similarly, $\forall\, p \in P,\ m^3(p) = m(p) - f(p, t_j) + f(t_j, p)$ and
   $m^4(p) = m^3(p) - f(p, t_q) + f(t_q, p)$.
   $\Rightarrow$ Combining equations for m^1, m^2 and m^3, m^4, we have
   $\forall\, p \in P,\ m^1(p) = m(p) - f(p, t_q) + f(t_q, p) - f(p, t_j) + f(t_j, p)$ and
   $m^4(p) = m(p) - f(p, t_j) + f(t_j, p) - f(p, t_q) + f(t_q, p)$.
   Then we have that $m^1 = m^4$, no matter the order chosen for the firing of transitions, the same marking is always reached.

   (b) Suppose $\exists\, t'$ such that $m[t_q t' t_j\rangle$ or $m[t_j t' t_q\rangle$. Without loss of generality let us assume $m[t_q t' t_j\rangle$.
   In this case, $m[t_q\rangle m''[t'\rangle$ and $m[t_q\rangle m''[t_j\rangle$. Note that we recursively analyze the case of transitions t', t_j as done for transitions t_q, t_j. Two possibilities can take place as transitions are fired:

      i. $|\mathcal{E}| \longrightarrow 0$ in which case the reached marking will always be the same, either because: the ordering does not matter (case 3.), an ordering is imposed in the transitions (case 2.), or because there is no sequence leading to a new consistent marking, then all possible sequences are rolled back to the last consistent state (Reduction rule 6.22).

      ii. $|\mathcal{E}| \longrightarrow \infty$, in which case no marking is ever reached for any firing order (we assume the firing of transitions is fair, so eventually, the transition making the firing sequence to diverge will fire). Thus, the "reached" marking is always the same (none).

3. $\circ t_q \cap t_j\bullet \neq \phi \vee t_q \bullet \cap \circ\, t_j \neq \phi$: Without loss of generality let us suppose that $\circ t_q \cap t_j\bullet \neq \phi$. If transition t_j fires before t_q, then t_q becomes disabled, because one of its inhibiting places contains a token.

   (a) Suppose that no internal transition t that becomes enabled, for which its firing removes the token from the inhibiting place of t_q. This means that when no other internal transition is enabled to fire, the CoPN has

an inconsistency. If this is the case, Reduction rule 6.21 is applied, and the state of the CoPN is reverted to marking m. Now transition t_q must be fired before t_j. The marking reached after all enabled internal transitions have fired, would be the marking m' of the CoPN.

(b) Suppose now that there is an internal transition t that consumes the token form the inhibiting place of t_q. Firing all enabled internal transitions would lead to a marking m' of the CoPN. Note that if t_q is fired before t_j the same marking m' would be reached. In particular, note that the transition will eventually be enabled, since transition t_j must fire as it is enabled. Consequently, the same sets of internal transitions will become enabled (probably in different order), and hence, the same marking m' would be reached.

The case in which transition t_q is fired before t_j is covered by the case (b). Hence, in this situation, the same marking is always reached.

□

Up until now, we have only dealt with CoPNs with a single priority for the set of internal transitions. Nonetheless, note that if future extensions require the definition of other transitions with different priorities, Theorem 6.3 remains valid. This is given because by definition transitions with different priorities are never enabled by the same marking.

6.4 Context-oriented Programming with Context Petri Nets

Now that the formal basis behind the structuring, definition and dynamics of COP systems has been described by means of the CoPN model, we turn to the programming support offered by the model. In this section we discuss the language support for the development of predictable COP systems using CoPNs and the definition of behavioral adaptations. Other aspects of the development of COP systems such as tool support are discussed in Chapter 8.

6.4.1 Language Abstractions for COP in CoPN

The CoPN model can become complex as the system grows. That is, the number of contexts and the number of interactions given by context dependency relations can yield a cluttered CoPN. However, developers interact with it through a language abstraction layer that hides such complexity. This section presents the API provided by the current incarnation of CoPN as a run-time model for Subjective-C. We explain the mapping of the different language constructs to CoPNs. It is important to notice that a similar API could be provided for different COP languages using CoPNs as a run-time model.

CoPN is a model to identify and manage inconsistencies in a COP language. Hence, the machinery introduced by CoPN to ensure system consistency and

Context declaration ::= **@context(** *context-name* [**,***bound*] **)**
Context activation ::=
    @activate(*context-name* [**in** *thread-name* {**,** *thread-name* }] **)**
Context deactivation ::=
    @deactivate(*context-name* [**in** *thread-name* {**,** *thread-name* }] **)**
Context method annotation ::= **@contexts** *context-name* { *context-name* }
Method priority declaration ::= **@priority** *priority*
ActivationState ::= **@active(** *context-name* **)**
Dependency relations declaration ::=
    [addExclusionBetween: *context-name* **and:** *context-name***]** |
    [addCausalityFrom: *context-name* **to:** *context-name* **]** |
    [addImplicationFrom: *context-name* **to:** *context-name* **]** |
    [addRequirementTo: *context-name* **of:** *context-name* **]** |
    [addSuggestionFrom: *context-name* **to:** *context-name* **]** |
    [addConjunctionOf: *{context-name}* **]** |
    [addDisjunctionOf: *{context-name}* **]**

Table 6.1: Subjective-C method syntax to interact with CoPNs.

behavioral predictability should not impact the way in which developers interact with the language. This is the case for Subjective-C. Table 6.1 shows the syntax for the creation and manipulation of contexts and context dependent behavior using CoPNs as the run-time model of the system, which resembles the one presented in Table 4.3 for Subjective-C. Note, however, that there are differences between the two syntax versions.

The context declaration construct now has the possibility to define a bound of the context (positive integers) as the maximum number of times a context can be activated. In the maps application, for example, in order to restrict the number of connections that the device can handle, the CONNECTIVITY context can be defined as `@context`(CONNECTIVITY, 3). Such a definition automatically generates a CoPN with an empty marking for a single context allowing a maximum of three simultaneous connections as shown in Figure 6.10.

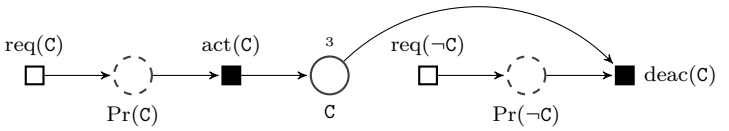


Figure 6.10: CoPN for the CONNECTIVITY (C) context allowing maximum three simultaneous activations.

The characteristic of enabling a bounded activation of contexts is a novel contribution to COP on its own. This characteristics addresses the multiplicity

problem identified in Table 4.2 not to be addressed by existing approaches, which provide either an unbounded activation of contexts, or a single activation of contexts.

Activation of contexts, remains unchanged. Section 9.2 further discusses the activation of contexts with a restricted scope for a particular thread. The context activation semantics introduced by CoPN differs from that of Subjective-C. Context activation `@activate(CONNECTIVITY)` and `@deactivate(CONNECTIVITY)` respectively trigger the firing of the external transitions req(CONNECTIVITY) and req(¬ CONNECTIVITY) of the CoPN in Figure 6.10. Section 8.1.3 Discusses the internals of context activation in CoPN and its difference with that originally proposed in Subjective-C.

Finally, the set of available context dependency relations is extended with the new *conjunction, disjunction* and *suggestion* dependency relations (the disjunction and suggestion dependency relations are defined in Section 9.3). The construct `addConjunctionOf:`, for example, takes a list of (an arbitrary number of) context names, and generates a CoPN of the conjunction of all of them (Definition 6.17). For example, the `[addConjunctionOf: [NSArray arrayWithObjects: CONNECTIVITY, FRIENDS,nil]]` construct generates the CoPN illustrated in Figure 6.7.

The DSL provided by Subjective-C to define contexts and context dependency relations of (Table 4.4) is extended as shown in Table 6.2 to use CoPNs as an underlying management model.

Context Declaration File	::=	`Contexts:` { *ContextName* } `Context dependency relations:` { *DependencyDefinitions* }
ContextName	::=	*context-name* [`,bound=` *number*]
DependencyDefinitions	::=	*ContextName DependencyConnector ContextName* | *ConjunctionConnector* | *DisjunctionConnector*
DependencyConnector	::=	*ExclusionConnector* | *CausalityConnector* | *ImplicationConnector* | *RequirementConnector* | *SuggestionConnector*
ExclusionConnector	::=	`><`
CausalityConnector	::=	`->`
ImplicationConnector	::=	`=>`
RequirementConnector	::=	`=<`
SuggestionConnector	::=	`-->`
DisjunctionConnector	::=	`+(` { *ContextName* } `)`
ConjunctionConnector	::=	`*(` { *ContextName* } `)`

Table 6.2: Subjective-C DSL syntax for CoPN.

As a concrete illustration, we round up this section with a complete example of the context-aware maps application (Section 2.3.3) developed using CoPN.

Example 6.5. The CoPN DSL definition of the maps application contexts and context dependency relations is shown in Snippet 6.1. Note that we left out the definition of the PRIV context for the management of user privacy in the application with the purpose of showing how contexts are added programmatically to an existing application, as shown on Line 1 of Snippet 6.2. To compose the

CoPN generated using the DSL with the `Priv` context it suffices to create a context dependency relation between such context and any of the other contexts already defined. This is done at Line 2 by creating an exclusion dependency between `Priv` and `Pos`.

```
Contexts:                    Context dependency relations:
    Pos                              W -> C
    N                                B -> C
    GPS                              GSM => Pos
    GSM                              GPS => Pos
    C,bound=3                        N => Pos
    W                                N =< C
    B,bound=1
```

Snippet 6.1: CoPN's DSL definition of the maps application.

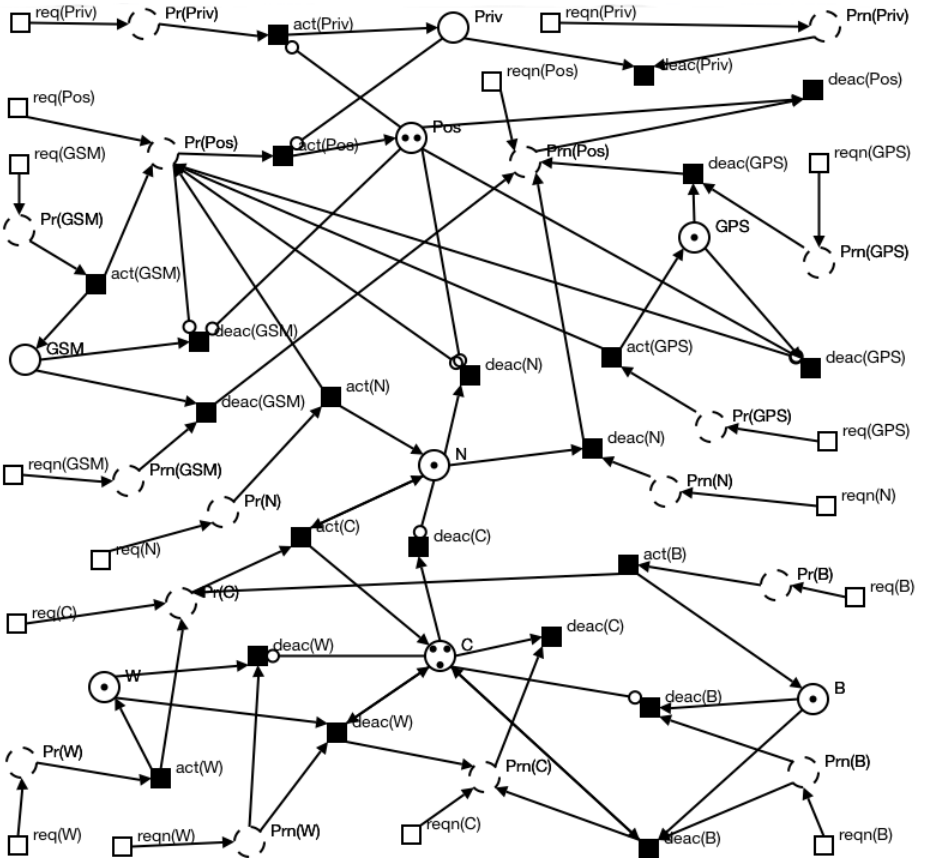


Figure 6.11: Composed CoPN for the maps application.

The CoPN shown in Figure 6.11 (automatically generated by the supporting tools introduced with CoPN, cf. Chapter 8) corresponds to the composition of all contexts defined for the maps application (this takes place automatically

after execution of Line 2 in Snippet 6.2).

```
1  SCContext *priv = @context(Priv,1);
2  [addExclusionBetween: priv and: @context(Pos)];
3
4  @activate(W);
5  if(!@active(C) ) {
6     @activate(C);
7  }
8  @activate(N);
9  @activate(Pos);
10 @activate(B);
11 @activate(C);
```

Snippet 6.2: Maps application: Context composition and activation.

The marking of the CoPN illustrated in Figure 6.11 is reached by executing the context activations shown on Lines 4 through 10 of Snippet 6.2. Line 4 requests the activation of the WIFI context. Due to the causality dependency relation W–▷C, a request to activate CONNECTIVITY is sent and both contexts are activated. The method @active(C) on Line 5 is used to check whether a context is active or not. In this case the return of the method is the boolean value *true*, since the CONNECTIVITY context is active. Line 8 request the activation of the NLBS context. Since NLBS requires CONNECTIVITY, N–◀C, and CONNECTIVITY is active the request can take place, so NLBS is activated. The consequence restriction given by the implication dependency relation N–▶POS, implies that the POSITIONING context is also activated. Line 6 requests the activation of CONNECTIVITY. This can immediately take place since CONNECTIVITY does not have any restrictions or consequences, leading to CONNECTIVITY to contain two tokens. Line 9 allows to activate POSITIONING since PRIVATE is not active, which is the only restriction to the context. Line 10 requests the activation of context BLUETHOOTH. This case is analogous to the activation of WIFI. This series of activations leads to the marking shown on the righthand side of Figure 6.11. Finally, Line 11 requests the activation of CONNECTIVITY, which cannot take place because the CONNECTIVITY context has reached its bound. An error message is sent providing the information: "Context CONNECTIVITY cannot be activated because it is preparing to activate and cannot complete the operation (CONNECTIVITY reached its bound)".

6.4.2 Context-dependent Behavior Semantics

The missing piece of the puzzle to provide a complete implementation of a COP language with CoPN, is the definition of behavioral adaptations associated with each context in the CoPN.

Before explaining how behavioral adaptations are represented in CoPN and how do they interact with the base system (i.e., the set of methods defined in the system for its common behavior) and other adaptations, we explain how the base system is represented in CoPN. In CoPN, the base system (its objects, state variables, and behavior) is represented by means of a special type of context, named the **default context**. This context is special in the sense that it cannot be manipulated by the programmer. The default context is a *system-*

defined context (much in the sense of the conjunction context generated by the conjunction dependency relation) which is *always* active. This representation is inspired by its initial proposal in the Ambience programming language [74] and also implemented in Subjective-C. In CoPN we abstract the default-context as a context only consisting of a context place which is always marked.

Definition 6.23 (**Default context**). *The default context is defined as the CoPN $\mathcal{C}^0$, where $P_c = \{\texttt{DEFAULT}\}$, $P_t = \phi$, $T_e = \phi$ and $T_i = \phi$, $f = \phi$, $f_\circ = \phi$, and $m_0(\texttt{DEFAULT}) = 1$. Moreover, $\forall m$ marking multiset, $m(\texttt{DEFAULT}) = 1$.*

In Chapter 4 we defined an adaptation which consists of two parts: a context describing a particular situation in the surrounding execution environment of the system, and a set of behavioral adaptations defined for that situation. Here we provide a formal definition of adaptations using CoPNs.

Definition 6.24 (**CoPN behavioral adaptations**). *In a COP system $\mathcal{P}$, an adaptation is defined as a tuple $< \mathcal{C}, \mathcal{B} >$ where $\mathcal{C} \in \mathcal{S}$ and $\mathcal{C} \subset \mathcal{P}$, and $\mathcal{B}$ is a set of behavioral adaptations (Definition 4.1) associated with the context place $p \in P_c$ in $\mathcal{C}$.*

Using Definitions 6.23 and 6.24, the base behavior of a COP system corresponds to the tuple $< \texttt{DEFAULT}, \mathcal{B}^0 >$, where the set $\mathcal{B}^0$ comprises *all* the methods defined in the base system.

In CoPN, behavioral adaptations are associated with a context by means of the `@contexts` annotation from Table 6.1, similar to the way it is done in Subjective-C. Methods defined in the application (be it in an object or in an independent module) can be annotated with the `@contexts` construct together with the name of the context the behavior is to be associated with. Snippet 6.3 shows the definition of the behavioral adaptation (`openChatRoom:`) associated with the FRIENDS+CONNECTIVITY context defined in the maps application. Note that writing `@contexts` FRIENDS+CONNECTIVITY (the context generated from the conjunction of the two contexts) is equivalent to the definition given on Line 1 in Snippet 6.3. That is, the behavior of `openChatRoom:` is made available if and only if both contexts FRIENDS and CONNECTIVITY are active, which corresponds to the semantics of the conjunction dependency relation.

```
1 @contexts Friends Connectivity
2 -(void) openChatRoom: {
3   ChatRoom *room = [[ChatRoom alloc] init];
4   if (connection != nil) {
5     friends = [self getOnlineFriends];
6     [chat start];
7   }
8 }
```

Snippet 6.3: Behavioral adaptation associated with context FRIENDS+CONNECTIVITY.

As contexts become available during the execution of the system, the observable behavior is provided by the sets $\mathcal{B}$ of behavioral adaptations for which their corresponding context place is marked together with the set of base level

methods $\mathcal{B}^0$. Problems may arise from the interaction of behavioral adaptations and the base level behavior, or even with other behavioral adaptations. If two behavioral adaptations corresponding to a same functionality are available, *what would be the behavior?*

To answer this question, selection and composition mechanisms for the resolution of methods are implemented, as discussed in Sections 4.1.2 and 4.1.4. We now provide a formalization of these concepts to avoid behavioral adaptations inconsistencies in CoPNs.

Definition 6.25. *Let $\mathcal{P}$ be a CoPN for a COP system with defined contexts $\mathcal{C}_1, \ldots \mathcal{C}_n \subset \mathcal{P}$. Given a method $b \in \mathcal{B}^0$, we define its equivalence class of b as, $[b] = \{b_i | b_i \in \mathcal{C}_i$ such that b_i is a behavioral adaptation of $b\}$.*

Definition 6.26 (**Method ordering**). *For every method b and for every $b_i \in [b]$, for $1 \leq i \leq n$, an ordering $\mathcal{O}$ on the equivalence class of b is defined as the sequence $b_{i_1} >_{\mathcal{O}} b_{i_2} >_{\mathcal{O}} \ldots >_{\mathcal{O}} b$, where the contexts $\mathcal{C}_{i_1}, \ldots$ are the active contexts in the system.*

The specific definition of the ordering $\mathcal{O}$ depends on the method disambiguation techniques (Section 4.1.4) implemented for each particular COP language. However, note that the ordering of the equivalence class is resolved dynamically according to the active contexts in the system. Furthermore, depending on the particular disambiguation techniques used, the ordering can be resolved for every method call.

Definition 6.27 (**Method resolution**). *Whenever a method b is called in the application, the method called is resolved by the behavioral adaptation b_i such that $b_i >_{\mathcal{O}} b_j$, $\forall\, b_j \in [b]$.*

Example 6.6. As a concrete example of a method ordering function $\mathcal{O}$, let us consider the method disambiguation technique implemented for the CoPNs programming model. CoPN reuses the method disambiguation techniques introduced in Subjective-C, *explicit priorities* and *activation order* (Section 4.3.3). To define the ordering and interaction of behavioral adaptations we introduce two predicates (one for each of the disambiguation techniques implemented) `priority` and `timestamp` which respectively retrieve the priority of a method and the timestamp of a context activation. Given two behavioral adaptations, b_i and b_j of a method b, according to the active contexts in the surrounding execution environment of the system, $b_i >_{\mathcal{O}} b_j$ in CoPNs if and only if:

1. $\texttt{priority}(b_i) > \texttt{priority}(b_j)$, or

2. $\texttt{priority}(b_i) = \texttt{priority}(b_j) \wedge \texttt{timestamp}(\mathcal{C}_i) < \texttt{timestamp}(\mathcal{C}_j)$.

As an example, consider the situation described in Snippet 6.4 for the maps application (Section 2.3.3), where the method `enableConnection` is defined for the two contexts `WLAN` and `Bluetooth`. In the situation of Line 8, where the two

contexts are available, which is the behavior that should be observed? According to Definition 6.27 using the two disambiguation techniques implemented with CoPN, two situations can take place. Let us suppose for the first case that the `@priority` annotation of Line 2 is not provided for any of the contexts. In such a case, since the `WLAN` context is activated later than `Bluetooth`, the observed behavior is the one associated with the `WLAN` context. The second case takes place when `@priority` is defined. In such a case, the observed behavior is that of the method annotated with the highest priority, the `Bluetooth` behavioral adaptation.

```
1  @contexts WLAN
2  @priority 10
3  -(void) enableConnection {
4    //connect via a WLAN
5  }

6  @activate(Bluetooth);
7  @activate(WLAN);
8  [self enableConnection];
```

```
@contexts Bluetooth
@priority 20
-(void) enableConnection {
  //connect via a Bluetooth
}
```

Snippet 6.4: Behavior composition and interaction in CoPN.

On top of the simple interaction of behavioral adaptations presented in this section, the CoPN programming model could also be further used for the composition, selection and scoping of adaptations. We discuss how the model could extended to provide explicit support for the dispatching of behavioral adaptations in Section 11.4.4.

6.5 Conclusion

This chapter provided the foundations for the development of a formal basis for the development of Dynamically Adaptive Software Systems based on the Petri net formalism. We presented context Petri nets (CoPNs) as a formal basis and programming model for the specific case of Context-Oriented Programming (COP) systems, and evaluate it with respect to the requirements of Dynamically Adaptive Software Systems (**D.1** – **D.5**) and run-time models (**M.1** – **M.4**). Satisfaction of these requirements is inherited from the combination of using COP systems (fulfilling **D.1**–**D.5**) and Petri nets (fulfilling **M.1**–**M.4**). In particular we highlight the following properties provided by CoPNs:

- CoPN provides a formal definition for the *structure and behavior* of COP systems. The CoPN model allows, unlike existing COP approaches, to formally describe the system in terms of its contexts, and the interaction between them. The dynamics of the system are defined as a formal semantics describing the activation of contexts (**M.3**).

- CoPN has a well-defined semantics describing the adaptations, context dependency relations, and composition of COP systems. The semantics

of the model are concise and precise, allowing us to define interaction between adaptation explicitly. Even more, the CoPN model manages implicit interaction of adaptation by means of its activation semantics. This reduces the amount of states required to represent the system and its adaptations (**M.1**).

- CoPN is used at run-time by the introduction of reactivity to external events. Events taking place in the surrounding execution environment are associated with the triggering of external transitions in the CoPN model. Every time there is a change in the execution environment, this is automatically represented in the CoPN model. The defined context dependency relations and activation semantics of CoPNs ensure at run time that the representation of the execution environment is always sound with respect to the expected system behavior (**D.1** and **D.4**).

- The CoPN programming model provides support for the principal language abstractions introduced by COP languages. As a consequence, CoPNs reuses these language facilities for the definition of behavioral adaptations. In particular, the model allows the definition of adaptations over different entities of a system (e.g., using the `@contexts` annotation), and to isolate such definitions from each other and the base system (e.g., using scoping of adaptations as will be explained later in Section 9.2). Furthermore, as behavioral adaptations are coupled with contexts, their introduction to and withdrawal from the base system are ensured to be consistent with respect to defined interactions between adaptations (**D.2** and **D.3**).

- CoPN provides an explicit representation of the data and control of the system at first-hand. Data, structure, interactions, and available actions of the system are all concisely represented in a single model, facilitating reasoning about the system (**M.4**).

The CoPN model provides the possibility of timely reaction to the surrounding execution environment of a software system in a consistent fashion. The model effectively provides support for the *interaction*, *abstraction*, and *decision* requirements for run-time models for Dynamically Adaptive Software Systems. Additionally, the model also provides a series of supporting tools, previously inexistent in other proposals for the definition and development of COP systems. However, we recognize that there is a price to pay for the additional support of ensuring consistency of the system behavior at run time. Every time there is a change in the surrounding execution environment of the system (a context activation or deactivation), the CoPN model is required to determine if such action conflicts with any of the other contexts available at that system state.

Chapter 7

Analyzing Dynamically Adaptive Software Systems

Chapter 6 presented and developed CoPNs, a formal basis for the definition and run-time management of behavioral adaptations upon contexts changes in the surrounding execution environment of the system. This chapter explores the reasoning power of CoPNs. We describe the design-time analysis of different properties of Dynamically Adaptive Software Systems using CoPNs, as a means to identify incoherences in the definition of interactions between adaptations.

In Chapter 6 we showed that every CoPN is guaranteed to have a consistent behavior with respect to the configuration of its adaptations as defined by context dependency relations. That is, given a specification of the system, expressed by context dependency relations and method resolution strategies, to guarantee that the specification is satisfied at run time. However, one question still remains: *how can we ensure that the defined context dependency relations define a valid specification of the system?*

In this chapter we address this question by developing the appropriate infrastructure to reason about COP systems. The idea is to complement the run-time resolution of conflicts between context dependency relations, with a design-time analysis process of the system properties. The reason to undertake such an analysis of system properties before its deployment is twofold. First of all, the questions that can be answered during the design of a system differ from those questions that can be verified at run time, thus improving the confidence and predictability of the system. Questions like: *will context* `A` *ever be activated?*, or *is activation of context* `A` *independent from that of context* `B`*?* are relevant to determine the validity of the system's specification. Secondly, a design-time analysis of the system could be used to lower the computational overhead of the run-time verification of the system's consistency by checking some of the properties beforehand.

The objective of analyzing system properties at design time is to inform programmers about potential behavior conflicts they had not foreseen. In this respect, CoPNs currently allow to reason about reachability and liveness properties of the system by taking advantage of existing Petri net analyses for such

properties. In this dissertation we use the LoLA tool to perform such analyses.

The following sections overview the process used in CoPNs to enable analysis of system properties. Section 7.1 presents different system properties that can be analyzed in Petri nets and their relevance in the setting of Dynamically Adaptive Software Systems. Section 7.2 presents a semiautomated process for the analysis of system properties using LoLA.

7.1 Reasoning About Systems Properties with CoPN

COP systems are guaranteed to behave consistently in the presence of context changes if they are specified using CoPNs. Specification of COP systems is driven by the definition of contexts and their interaction using context dependency relations —that is, the composition of the defined contexts in the system using the composition ($\circ$) operator introduced in Definition 6.6. The composition operator is well defined in the sense that the composition of two consistent COP systems always yields a consistent COP system. The resulting composed CoPN, however, may not always yield a *coherent* system.

Definition 7.1. *For a given CoPN* $\mathcal{P} = \langle P_c, P_t, T_e, T_i, f, f_\circ, \rho, \mathcal{L}, m_0, \Sigma, \lambda \rangle$*, let* $p \in P_c$*, we define* p *to be* ***unreachable*** *if* $\forall\, m$ *reachable marking multiset of* $\mathcal{P}$ *from the initial marking* m_0*,* $m(p) = 0$*. Context places* p' *for which there exist a marking multiset* m *such that* $m(p') > 0$ *are called reachable.*

The notion introduced by Definition 7.1 corresponds to the reachability property of Petri nets (Definition 5.6). This notion is used throughout this chapter to verify the coherence of CoPNs by reasoning about the activation state of a subset of the contexts defined in the system.

Remark Since Theorem 6.2 states that every finite step yields a consistent state, it can be derived from Definition 6.19 that if Υ is unstable, there is an infinite number of internal transitions in the firing step of Υ. Finite steps are called stable.

Definition 7.2 (Coherence). *A CoPN* $\mathcal{P} = \langle P_c, P_t, T_e, T_i, f, f_\circ, \rho, \mathcal{L}, m_0, \Sigma, \lambda \rangle$ *is coherent if* $\forall p \in P_c$*,* p *is reachable, and* $\forall\, t \in T_e$*, the firing steps* Υ *with* t *as the first fired transition are stable. A CoPN that is not coherent is called incoherent.*

Note that the run-time verification process to ensure consistency of a CoPN when contexts are activated and deactivated is not sufficient for the identification (and eventual avoidance) of *incoherences*. In particular, at run time, it is not possible to determine if a step is unstable or not (determining this is equivalent to the halting problem). However it is possible to identify incoherences by analyzing the system statically at design time. To avoid incoherences in the composed CoPNs, we use the decision tools of Petri nets as part of the CoPN model. The CoPN model does not provide dedicated techniques for the analysis of COP systems *per se*. Instead, it uses the different analyses of Petri net

properties already existing. The properties that are interesting to be analyzed in the setting of COP systems are:

Reachability: In the case of COP systems, analyzing the reachability and coverability of a set of states can be seen as the possibility of identifying whether a particular configuration of active contexts is possible (a set of simultaneously marked context places), given a system state (an initial marking of the corresponding CoPN). This property can be used to identify if a given context can be activated, or to verify if two contexts can be simultaneously active when they should not. For example, to verify whether two contexts in an exclusion dependency relation could ever be active at the same time.

Liveness: In the case of COP systems, analyzing the liveness property of the system (in its stronger version L_4 defined in Definition 5.7) can be seen as identifying if a particular action over a context could ever take place (if internal transitions adjacent to context places would ever fire), given a system state (an initial marking for the CoPN). For example, liveness could be used to identify if the deactivation transition of the source context in a requirement relation could fire or not.

Persistence: In the case of COP systems, analyzing the persistence property of the system can be seen as identifying disconnected components of the CoPN —that is, sets of contexts which only have context dependency relations between each other, and have no context dependency relation with any other contexts outside the set. This property can be used to better modularize features of the system, or to identify undesired independence of contexts.

Fairness: In the case of COP systems, analyzing the fairness property of the system can be seen as identifying contexts whose activation can never occur, because its associated transitions cannot fire in a step. Such a situation occurs when other transitions in the step fire infinitely (their internal transitions starve for the given marking for the CoPN). This property can be used to identify situations in which the system would become irresponsive. For example, to identify the problem of the implication dependency relation deactivation cycle (Example 4.5).

As the number of contexts and context dependency relations defined in a COP system increases, incoherent CoPNs could be created. The composition of CoPNs and COP systems in general should be analyzed for incoherences. Each of the following examples hints at how the analysis of Petri net properties could be used to identify these problematic situations.

Example 7.1. The first example of an incoherent CoPN produced by composing simpler CoPNs is given by the composition of two causality dependency relations, $\mathtt{A}\text{–}\triangleright\mathtt{B} \circ \mathtt{B}\text{–}\triangleright\mathtt{A}$, as shown in Figure 7.1. The incoherence in the composed CoPN

is given by an infinite loop between the activation transition of context A and the activation transition of context B, which is depicted in the figure by the arcs in bold. In this case, the CoPN presents an accidental interaction between contexts A and B. This interaction is the reason for the system to block (and eventually crash) every time one of the two contexts is requested for activation. The interaction between the two activation transitions constitutes an example of a CoPN in which a step is unstable.

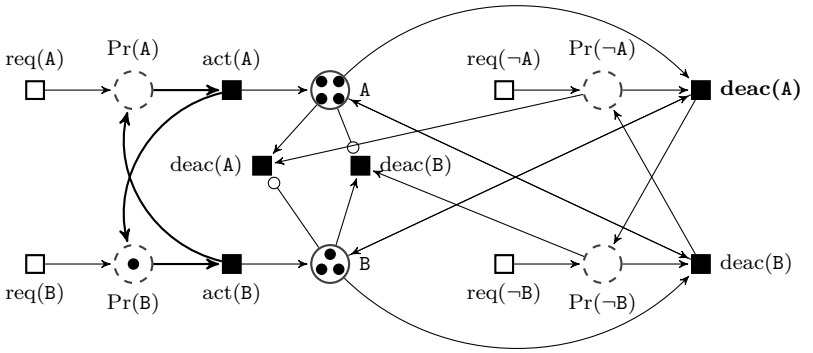


Figure 7.1: A CoPN with an unstable step for the activation of its contexts.

Beginning with an empty marking of the system (none of the two contexts is active), the request for activation of A through the firing of the req(A) external transition adds a token to the Pr(A) temporary place. This new marking m enables the act(A) internal transition. Firing of the internal transition respectively marks places A and Pr(B), enabling transition act(B). When this transition fires, it adds a token to places B and Pr(A). The state of the CoPN at this point subsumes the state generated immediately after the firing of the req(A) external transition. Moreover, since no other internal transition is enabled in the transition firing sequence, the same sequence would occur again. That is:

$$\forall\, m' \text{ such that } m[\Upsilon\rangle m',\ \text{we can derive that } \forall\, p \in P\ m'(p) \geq m(p)$$

The state of the system shown in Figure 7.1 is reached after 3 more firings of the act(A) transition. The step started by firing req(A) is unstable because at every reachable state there is always an internal transition to be fired (act(A) or act(B)). This would cause the system to block indefinitely, or even to crash.

In case one of the two contexts is bounded, the firing step is stable. The activation process goes on as before until the bound of the context is reached. Whenever this happens an error message is signaled and the activation is rolled back. This error is signaled because there is always one extra request for the activation of the context after its bound has been reached. As a consequence, every activation request for context A or B would raise an error, so none of the contexts can ever be activated. The inconsistency shown in Example 4.5 for the

deactivation cycle problem of Subjective-C is similar to the situation described in this example.

Incoherences as the one shown in this example, could be identified by analyzing properties of the composed CoPN. Fairness could be used to check if the rightmost transition deac(A) (bold in Figure 7.1) could ever be fired. A test for fairness would evidence that both act(B) and act(A) transitions would fire infinitively many times, disallowing any other transition of being fired. Other properties that could be tested include liveness (of transition deac(A)), or reachability of a marking m, where $m(\mathtt{A}) = 1$ and $m(\mathtt{B}) = 1$ and no other place is marked. Such a marking is not reachable.

Example 7.2. A second example of an incoherent CoPN produced by composing three coherent CoPNs is that of composing a causality, an implication, and an exclusion dependency relations, A–▷B ∘ A–▶C ∘ C□–□B. Figure 7.2 shows the composed CoPN. The incoherence in this case is given by context A being unreachable. The CoPN presents an accidental interaction between the three contexts. Such interaction causes the system to raise an error every time context A is requested for activation. Nonetheless, it is still possible to activate contexts B or C independently. The interaction between the three contexts and their context dependency relations constitutes an example of adaptations that cannot occur simultaneously, even if they could from their individual context dependency relation definitions, for example, context A and C could be activated if B is inactive.

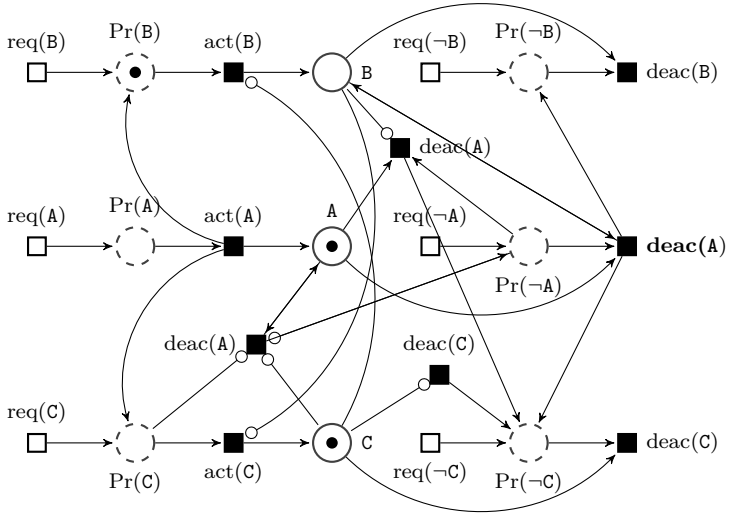


Figure 7.2: A CoPN with non-reachable context A.

To demonstrate the interaction between the contexts let us suppose that we

beginning with an empty marking of the system. A request for activation of context A triggers the firing of the req(A) external transition, which adds a token to Pr(A). This marking enables the act(A) internal transition. Firing this transition adds a token to the temporary places Pr(B) and Pr(C). In this marking the two internal transitions act(B) and act(C) are enabled. Since the two transitions have the same priority, they fire randomly. Without loss of generality, suppose act(C) is fired, yielding a marking of context places C, A and the temporary place Pr(B), as shown in Figure 7.2. However, due to the exclusion dependency relation between contexts C and B, transition act(B) is no longer enabled and cannot fire. In fact, in this marking no internal transition is enabled. The CoPN is in an inconsistent state because temporary place Pr(B) is marked. As a result, the activation of context A raises an error and it is rolled back .[1]In this situation, the system does not have any perceivable errors. Nonetheless, part of its behavioral adaptations can never take place, even though the situations in the surrounding execution environment render them more appropriate. The interaction between the contexts prevents the system of behaving as predicted. Even if context A is available in the surrounding execution environment, the behavioral adaptations associated with the context are never selected as part of the system's behavior.

Incoherences as the one exposed in this example, could be identified by analyzing properties of the composed CoPN. Reachability could be used to check if context activation takes place as expected, inspired by the expected interaction from the independent context dependency relations. That is, if it is possible to reach situations in which context A and B are active, or contexts A and C are active. Liveness could also be used to ask if the internal transition deac(A) (bold in Figure 7.2) would ever fire.

The following sections provide the details on the analysis process of reachability and liveness in CoPNs, driven by the examples of the possible incoherences that could be presented through the composition of CoPNs.

7.1.1 Generating CoPNs with Place Capacities

The analysis of Petri net properties is a challenging problem, even more so when we leave the realms of standard Petri nets. A large body of research is dedicated to studying the theoretical aspects of the decidability of Petri net properties over different classes of Petri nets [12, 55, 113, 54], as well as the development of techniques and algorithms that effectively analyze a particular property [168, 116, 155, 95, 139].

In the general case, analysis of Petri net properties is undecidable in the presence of inhibitor arcs, due to the fact that Petri nets with inhibitor arcs are equivalent to Turing machines. Deciding the reachability problem for Petri nets with inhibitor arcs would be equivalent to deciding the halting problem, which is proven to be undecidable. It has been proven, however, that the problem

[1]The reasoning for firing act(B) is similar.

is decidable for particular types of Petri nets with inhibitor arcs. Examples of these are: (1) the class of Petri nets with only one inhibitor arc [155], (2) the class of Petri nets in which an ordering on its places can be defined such that a place is an inhibiting place of a transition, if every other inhibiting place is a preceding place in the order [155], (3) the class of bounded Petri nets, where the Petri net can be unfolded into an equivalent Petri net without inhibitor arcs, and (4) the class of *primitive systems*,[2] in which case the Petri net may be unfolded to an equivalent Petri net without inhibitor arcs [26].

Decidability of property analyses for CoPNs becomes problematic when the model does not adhere to any of the aforementioned cases of Petri nets. CoPNs may have multiple inhibitor arcs through the composition of COP systems satisfying more than one context dependency relation. The composition of CoPNs also makes it difficult to define an ordering on the Petri net places such that inhibiting places for a transition come from a preceding place in that order. Inhibiting places may comply to the ordering condition of each individual CoPN, but not necessarily to the order in the composed CoPN. CoPNs are in principle unbounded, but they can be bounded according to the specification of the system domain. However, in the general case, it may be unfeasible to set a bound for all contexts, for example, in the setting of an algorithm optimization system through parallelization, where a `CONCURRENCY` context is activated as many times as new execution threads are spanned in the system. The definition of the `CONCURRENCY` context cannot be bounded *per se*. CoPNs do not comply with primitive systems, as inhibiting places could always be emptied, for example, by inactivating a context.

In order to enable Petri net analyses on CoPNs we therefore restrict the specification of the system to bounded CoPNs, where a bound is given to every place. It comes as no surprise that bounding a CoPN restricts its semantics. In a CoPN with place capacities contexts are not allowed to be freely activated because contexts can only be activated up to their given bound. The consequence of bounding the places in CoPNs, is that some of the consistent firing steps may become inconsistent. This is due to the restriction for firing internal transitions when an output place of the transition has reached its bound. However, taking into account the type of required analyses from Examples 7.1 and 7.2, we observe that most of the interesting analyses to be performed refer to particular states being active or not —that is, not minding how many times a context has been activated. Thus it is still possible to extract valuable information from the analysis of bounded CoPNs as shown in this chapter.

The process of unfolding a CoPN into a Petri net with place capacities and without inhibitor arcs follows the ideas of the unfolding algorithm for primitive systems into equivalent Petri nets provided by Busi [26]. However the semantics of the unfolded CoPN is not equivalent to that of the original CoPN.

The first step to unfold a CoPN is to set a bound for all its places. A larger value for the bound would generate a larger number of places and transitions.

[2] Primitive systems [26] are a class of Petri nets with inhibitor arcs, in which inhibiting places can be guaranteed never to be emptied once they have reached a certain capacity.

This implies that the state space to be analyzed by the different analyses would be larger. For analyzing the system's properties we use the Low Level Petri net Analyzer (LoLA) tool [169]. We provide a bound k to each place in the CoPN that complies with the size of the state space that LoLA can handle while keeping it as large as possible so that the semantics of the CoPN are preserved as far as possible. The process of choosing such a bound for each place is explained later in this section.

Snippet 7.1 shows our unfolding algorithm which consists of replacing each inhibiting place for a set of places, where each place in the set represents the number of tokens contained in the original place (including having no tokens). Each set of places is thus of size $k + 1$. Transitions with input inhibitor arcs are replaced by a set of transitions (of size equal to the greatest bound of its inhibiting places), where each transition is incident to two places from the generated set of places (associated to one of the original places).

```
1   loop for p ∈ P such that p is inhibitor
2     let k a bound of p
3     loop for i from 0 to k
4       add new place p^i
5     loop for t such that p ∈ t• or p ∈ •t
6       if p ∈ ∘t then
7         replace arc(p,t) with new arc(p^0,t) and new arc(t,p^0)
8       if p ∈ t• and p ∉ •tthen
9         loop for i from 0 to k − 1
10          add new transition t^i
11          add new arc(p^i,t^i)
12          add new arc(t^i,p^(i+1))
13      if p ∈ •t and p ∉ t•then
14        loop for j from 1 to k
15          add new transition t^i
16          add new arc(p^i,t^i)
17          add new arc(t^i,p^(i−1))
18      if p ∈ t• ∧ p ∈ •t then
19        loop for i from 1 to k
20          add new transition t^i
21          add new arc(p^i,t^i)
22          add new arc(t^i,p^i)
```

Snippet 7.1: Unfolding algorithm of CoPNs into bounded CoPNs without inhibitor arcs.

The algorithm depicted in Snippet 7.1 is described as follows:

1. We define a capacity k (not necessarily the same) for all places in the Petri net (Line 2).

2. Every inhibiting place p (a place such that $p \in \circ t$ for some transition t), is replaced by a set of places $\{p^i | i = 0, \ldots, k\}$ (Lines 3 to 4).

3. Inhibitor arcs to the transition t are replaced by the arcs (p^0, t) and (t, p^0) (Lines 6 and 7).

4. Furthermore, each transition t incident to p is replaced by a set of transitions, each of which manages a specific representation of the contents of place p by means of places p^i (Lines 5 to 22):

a) If $p \in t\bullet$ then t becomes the set $\{t^i | i = 0, \ldots, k-1\}$ (Line 8). When t^i fires it removes a token from p^i and adds a token to place p^{i+1} (Lines 11 and 12).

b) If $p \in \bullet t$, then t becomes the set $\{t^i | i = 1, \ldots, k\}$ (Line 13). When t^i fires, a token is removed from p^i and added to p^{i-1} (Lines 16 and 17).

c) If $p \in t \bullet \wedge p \in \bullet t$ then t becomes the set $\{t^i | i = 1, \ldots, k\}$ (Line 18). t^i can fire only if place p^i is marked. Its firing does not modify the state of the p^i (Lines 21 and 22).

5. All original input and output places of each transition remain as they were originally (except for those modified in the algorithm). Similarly, transitions not incident to any inhibiting place remain as in the original CoPN.

In order to define the bound k for the CoPN places, note that the number of places and transitions in a CoPN depends on the number of contexts and context dependency relations defined in it. Additionally, remember from Section 5.1.3 that the maximum size of Petri nets that can be handled by LoLA is on average of 500 places and 1000 transitions. Choosing a bound requires that the state expansion of the generated sets of places and transitions by the algorithm does not go over such limits. We now explain how bounds are chosen for CoPN according to the limits of the state space size handled by LoLA.

Given a CoPN $\mathcal{P} = \langle P_c, P_t, T_e, T_i, f, f_\circ, \rho, \mathcal{L}, m_0, \Sigma, \lambda \rangle$. Let $I \subseteq P$ be the set of inhibiting places of $\mathcal{P}$, and $IT = \{t \,|\, \exists\, p \in I \text{ and } p \in \bullet t \vee p \in t \bullet \vee p \in \circ t\}$ be the set of transitions incident to inhibiting places in $\mathcal{P}$. The corresponding bounded Petri net without inhibitor arcs of $\mathcal{P}$ is defined as $\overline{\mathcal{P}} = \langle \overline{P}, \overline{T}, f, \rho, \overline{m_0} \rangle$, where $\overline{P} = (P \setminus I) \cup I^{k+1}$, $\overline{T} = (T \setminus IT) \cup IT^k$ and $\overline{m_0}$ is the restricted marking multiset to the bound of each place. The sets I^{k+1} and IT^k, respectively are the sets of generated sets of places and transitions by the algorithm of Snippet 7.1. Let us suppose that $|I| = n$, and $|IT| = q$, then the state space of the generated bounded Petri net without inhibitor arcs is restricted by:

$$|\overline{T}| = |T| + q(k-1) \leq 1000 \tag{7.1}$$

$$|\overline{P}| = |P| + nk \leq 500 \tag{7.2}$$

From Equations 7.1 and 7.2, is possible to see in Equations 7.3 and 7.4 that the bound of the Petri net is the ratio of the total number of places/transitions to number of the inhibiting places/transitions with inhibitor arcs.

$$k \leq \frac{1000 - |T|}{q} + 1 \tag{7.3}$$

$$k \leq \frac{500 - |P|}{n} \tag{7.4}$$

Equations 7.3 and 7.4 provide an approximation for the maximum bound for a CoPN for which the state expansion can be managed by LoLA. However, this bound is not satisfactory for all places. Remember from Table 6.1 that contexts

can be given a bound, if so required by the application domain. Whenever places are given a bound, it should be preserved to ensure that the semantics of such a context is not lost.

To calculate the bound of every place in a CoPN, the bounds defined for the original CoPN are reused. Each place that initial had no capacity is given as a capacity the minimum between its the generated capacities by CoPNs (Equations 7.3 or 7.4). Every place with an initial capacity is given a capacity its original bound in case that this is smaller than the generated ones. If the initial capacity of the place is bigger than the generated capacities, then the place is given the maximum between the generated capacities. Equation (7.5) shows the definition of the capacity k_p given for a place p.

$$k_p = \begin{cases} \min\left\{\dfrac{1000-|T|}{q}+1, \dfrac{500-|P|}{n}\right\} & \text{if } \nexists\, k_0 \text{ for place } p \\ k_0 & \text{if } \exists\, k_0 \text{ for place } p \\ \max\left\{\dfrac{1000-|T|}{q}+1, \dfrac{500-|P|}{n}\right\} & \text{if } k_0 > (7.3) \wedge (7.4) \end{cases} \tag{7.5}$$

Example 7.3. As an example of how these formulae are used to calculate concrete capacities for a CoPN, let us revisit the CoPNs described in Figures 7.1 and 7.2. Suppose that none of the places in the CoPNs have been given a bound. In the case of the CoPN of Figure 7.1 $|P| = 6$, $|T| = 10$, $|I| = 2$, and $|IT| = 6$. From Equation (7.3) we have that $k \leq 170$, and from Equation (7.4) $k \leq 247$. The chosen bound for every place is 170. Similarly, in the case of the CoPN of Figure 7.2, $|P| = 9$, $|T| = 16$, $|I| = 3$, and $|IT| = 7$. From Equation (7.3) we have $k \leq 141$, and from Equation (7.4) we have $k \leq 162$. The chosen bound for every place is 141.

Suppose that we give a bound of 2 for every context place in Figures 7.1 and 7.2. according to Equation (7.5), context places will maintain their bound of 2, while temporary places will be given the generated bound, 170 and 141, respectively.

Example 7.4. This example shows the unfolding of the CoPN of Figure 7.1. Let us suppose we define the contexts of the CoPN as bounded contexts, where context `A` is given a bound of 1, and context `B` is given a bound of 2. Further suppose that the initial state of the CoPN is an empty marking. The bound for the temporary places (originally unbounded) is then $k = 170$. The corresponding Petri net with place capacities and without inhibitor arcs is shown in Figure 7.3. For the sake of simplicity Figure 7.3 does not show the bound assigned to initially unbounded places, all places not labeled with a bound are assumed to have a bound of 170.

Notation 7.1.

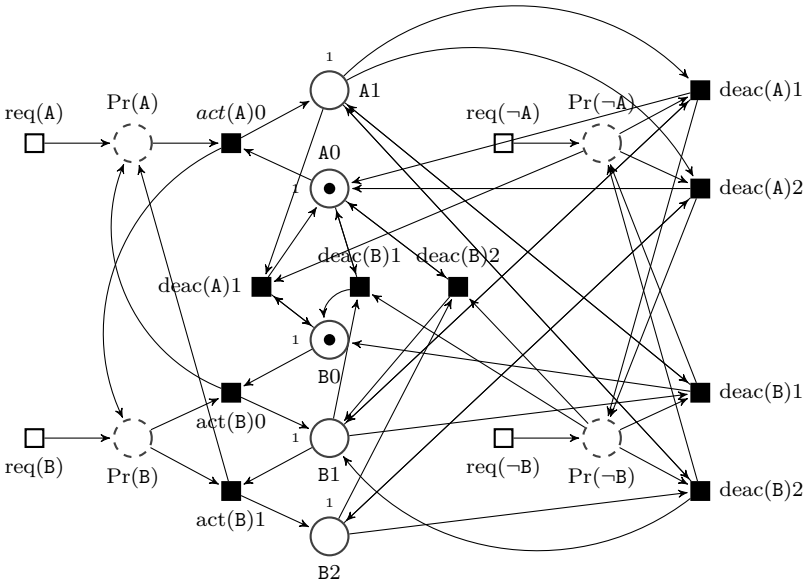


Figure 7.3: Petri net generated from the unfolding of the CoPN $\mathcal{P} = \circ(\{\mathcal{C}_A, \mathcal{C}_B\}, \{\langle C, \mathcal{C}_A, \mathcal{C}_B\rangle, \langle C, \mathcal{C}_B, \mathcal{C}_A\rangle\})$.

- *In unfolded CoPNs elements are labeled using the label corresponding to the original element appended by the number of their input bounded place. For example, a transition labeled deac(*A*)1 removes the token of place* A*1 and adds a token to the place* A*0*

The process of bounding CoPNs preserves the semantics of reactive Petri nets and Petri nets with static priorities. However, these two semantics cannot be used when analyzing the CoPN with LoLA. Reactive Petri nets, or Petri nets with static priorities are not among the family of high-level Petri nets supported by LoLA. Nonetheless, the generated Petri net can be stripped of these semantics using the results of Theorems 5.1 and 5.2. The generated Petri net can be treated as a low-level place/transition net —that is, a Petri net with the regular token-game semantics, where for every valid firing sequence of transitions σ under the regular semantics, a permutation of σ leads to a consistent step under the CoPN semantics. Furthermore, every accepted sequence of transition firing in the generated Petri net without inhibitor arcs is also a consistent sequence of steps in the original CoPN.

7.1.2 Analysis of CoPN Properties

Once a CoPN has been unfolded to a bounded Petri net without inhibitor arcs, it can be used as input for a Petri net analyzer, such as LoLA [169]. Using LoLA, we are able to analyze the coherence of (a restricted semantics of) the system

according to Definition 7.2. In particular, we analyze whether contexts can ever be active simultaneously (reachability, as in Definition 7.1), if a context can ever be activated (liveness), if there are configurations of contexts in which no action can be taken (deadlocks), and other relevant states of the system. We explain each of these analyses in more detail by using the examples of Figures 7.1 and 7.2. Section 9.1 validates the analysis process of CoPNs using a larger case study.

Reachability

Reachability can be used in the setting of COP systems to reason about the states of contexts. A reachability analysis can verify positive and negative properties of the system. That is, examining the state of contexts to verify that their interaction takes place as expected. For example, in the case of an implication dependency relation, to verify that every time the source context is activated, the target context is also activated. Moreover, reachability analysis can also be used to check that interaction constraints are satisfied, for example, that two contexts are never active at the same time in an exclusion dependency relation.

Testing reachability with LoLA requires the (LoLA formatted text) representation of the Petri net, as the one shown in Snippet 5.1, and the target state of the Petri net, that is, the state we are analyzing. The analysis property is formatted in LoLA as shown in Snippet 7.2, where the analysis is described as an analysis over the whole Petri net, giving the final expected marking. This definition is used to identify if, for example, it is possible to reach a marking in which the two contexts `A` and `C` are simultaneously active.

```
//Is there a state where A and C are marked?
ANALYSE MARKING A:1, C:1
```

Snippet 7.2: LoLA reachability analysis.

Running such an analysis with LoLA generates two kinds of outputs. On the one hand, it provides an output value, denoting whether or not the run was successful, in this case, whether the marking with only the two context places containing one token was reached. On the other hand, it generates two output format files, namely the state and path output files. The state output file consists of the final marking of the system after the analysis was performed, which can be used to observe if the reached state is effectively as expected —no extra or missing tokens. The path output file consists of the firing sequence of transitions leading to the desired state. This information can be further analyzed by the developer to see whether the desired state could be reached through a consistent step in the regular CoPN semantics. It is possible that the generated marking of the analysis subsumes the desired state. Although LoLA gives a negative result, the reached marking could still be acceptable if, in addition to `A` and `C`, some other contexts places are marked as well, and no undesired transitions where fired to reach such a state. Such a situation could

take place, for example, when the initial marking of the system is not the empty marking.

Example 7.5. Remember from Example 7.2 that we are interested in finding out if it is possible to activate context A. Given the context dependency relations through which context A interacts, activation of context A has as a consequence the activation of contexts B and C, respectively A–▷B and A–▶C. Let us assume a bound of 2 for each of the contexts B and C and a bound of 1 for context A to simplify the analysis of the example. We are interested in verifying whether the state given (in the unfolded CoPN) by $m(\texttt{A1}) = 1$, $m(\texttt{B1}) = 1$, $m(\texttt{C1}) = 1$, and $m(Pr(\texttt{C})0) = 1$ is reachable or not.

Note that regardless of the chosen value of the bound k, in order to check if the contexts are active at the same time, it is only necessary to check if the places marked with one token is reachable (e.g., $m(\texttt{A1}) = 1$, $m(\texttt{B1}) = 1$, $m(\texttt{C1}) = 1$, $m(Pr(\texttt{C})0) = 1$). If this is the case, it is not necessary to check the various other cases, for example, $m'(\texttt{A}) = 1$, $m'(\texttt{B2}) = 1$, $m'(\texttt{C1}) = 1$, and $m'(Pr(\texttt{C})0) = 1$, since to reach these markings it is necessary to obtain marking m first.

As indicated by LoLA the desired state m is never reached. The information provided by this simple analysis signals a problem in the definition of the context dependency relations. However, the provided information is not enough to know what the problem is. Performing another reachability analysis on the exclusion dependency relation defined between contexts B and C indicates that it is not possible to reach a state $m''(\texttt{C1}) = 1$, $m''(\texttt{B1}) = 1$, and $m''(Pr(\texttt{C})0) = 1$ (as expected). With this additional analysis it is possible to know that the three defined context dependency relations in the system interact accidentally not allowing the activation of context A.

Example 7.5 shows how to use the LoLA reachability analysis and to reason about its results in the setting of CoPNs. However, in order to reason about the availability of context A we did not test the state of this context on its own. Rather, we tested for the availability of all three contexts. The choice of analyzing the state of the three contexts is motivated by two reasons. First, context A has two context dependency relations that constrain the activation of context A to always activate contexts B and C, respectively. Second, we are interested in reasoning about consistent firing steps in the CoPN —that is, steps between firings of two external transitions, since this is the semantics used at run time. The reachability of a state where A is marked is not allowed under CoPN semantics, since there remain enabled internal markings to be fired. Section 7.2 explains the process of generating the required test cases to analyze system properties for the context dependency relations.

Liveness & Deadlocks

Liveness can be used in the setting of COP systems to reason about whether an action is executable in the system. For example, to reason about the possibility

of firing the deactivation transition of a context. Liveness is closely related to the test for deadlocks, where the Petri net is analyzed to check if there are system states for which no action can take place. In the setting of COP systems deadlocks refer to particular context configurations for which the system cannot adapt its behavior anymore as adaptations cannot be activated.

Reasoning about liveness is an expensive operation and in LoLA liveness analysis is provided as a local property —that is, transitions need to be analyzed individually for liveness. To verify the liveness property of a transition, an input analysis file as shown in Snippet 7.3 must be used as input for the LoLA analysis.

```
//Is transition deac(A) dead?
ANALYSE TRANSITION deac(A)
```

Snippet 7.3: LoLA liveness analysis.

LoLA also allows us to reason globally about deadlocks in the Petri net — that is, reachable states in which transitions are no longer fireable. Deadlock analysis in LoLA does not require an input analysis file.

Running a liveness or deadlock analysis provides two kinds of outputs. As for the reachability analysis an output value is generated denoting whether the transition is live or not (in the case of liveness) or whether there are dead transitions (in the case of deadlocks). Additionally, when verifying a transition for liveness, the tool can output the state in which the transition is not dead, and the sequence of transition firings leading to that state. In the case of deadlock analysis, if a dead state is found (a state in which dead transitions exist) the path leading to such a state is provided as output.

Example 7.6. Remember from Example 7.1, that we are interested in finding out if contexts A and B could be active at the same time, as the intuition of the causality dependency relation would suggest. Activation of one context automatically triggers the activation of the other. Given the initial marking as shown in Figure 7.3 we want to know if there is a state in which the rightmost deac(A) transition is dead or not.

Analyzing the unfolded CoPN by means of a query similar to that shown in Snippet 7.3 shows that the transition is indeed live. However, taking a closer look at the state and path outputs, it is possible to see that the state in which the transition is not dead is given by, for example, a marking $m(Pr(\mathtt{A})) = 1, m(\mathtt{A1}) = 1, m(\mathtt{B}) = 1$, $m(Pr(\neg\mathtt{A})) = 1$. Moreover, the sequence of fired transitions to reach the state is $\{req(\mathtt{A}), act(\mathtt{A})0, act(\mathtt{B})0, req(\neg\mathtt{A})\}$.[3] From the two outputs we can derive that the (internal) transition act(A)0 is enabled and has to fire again before the firing of req(¬ A). Hence, hinting at the existence of a loop in the activation of the contexts A and B. Note also that the sequence of transitions given by the tool does not reaches of a consistent step. As a result we can conclude that the state in which deac(A)1 is live, is not actually

[3]This is the sequence of fired transitions modulo permutation of transitions of the same priority. Other sequences reaching the same state could be given as output.

reachable in the CoPNs semantics. This means the CoPN is incoherent, since there is an unstable step.

The analysis of the output file in the case of liveness requires more involvement from part of the programmer than that of the reachability analysis. Even though a transition might be live in the regular token-game semantics this might not be the case in the CoPN semantics, for example, because a transition with higher priority is enabled. Therefore, each of the produced sequences must be carefully verified to see if they can take place under the CoPN semantics. An automated verification of such output paths could be implemented by analyzing the sequences of transition firings. This process is discussed as future work in Section 11.4.4.

State predicates

State predicate analysis can be used in the setting of COP systems to reason about states or state invariants that should be fulfilled by the system. State predicates can be used to reason about other Petri net properties, such as reachability or liveness, whenever these are difficult to express or do not yield any result. Examples of state properties to test for reachability and liveness are shown in Snippet 7.4. Line 2 tests if it is possible to reach an state in which the marking of `B` is smaller than the marking of `A` (such state should not be reached). Line 4 tests for the liveness of the deac(`A`) transition in Figure 7.1, by checking if the the state enabling the transition can be reached.

These predicates respectively test a particular state of the system, and the state enabling a particular transition.

```
1 //Marking for context A is always smaller than that of context B?
2 FORMULA B = 0 AND A > 1
3 //Can transition deac(A) be fired?
4 FORMULA Pr(¬ A)>0 AND B>0 AND A>0
```

Snippet 7.4: LoLA liveness analysis.

If a state matching the state described by the state predicate formula is found, the analysis provides the reached state as output. However, unlike the tests for reachability and liveness, the path to reach such a state is not provided. It is up to the programmer to deduce any additional information required from this state output.

Example 7.7. The conditions verified by the reachability property in Example 7.5 for the CoPN of Figure 7.2 could also be verified by means of state predicates. The state formula would resemble that of Snippet 7.2 indicating that both `A1` and `C1` should have a marking greater than zero.

```
FORMULA A1 > 0 AND C1 > 0
```

LoLA cannot reach the state specified by this formula. Taking a closer look at the corresponding CoPN we note that whenever contexts `A` and `C` are marked,

so is place Pr(B), as shown in Figure 7.2. Thus it is not possible to reach a state where only A and C are marked.

Example 7.8. The liveness property of the deactivation transition deac(A) of Example 7.6 could also be verified by means of state predicates. To verify if the rightmost transition deac(A)1 in Figure 7.3 is live or not, the predicate formula in Line 4 of Snippet 7.4 could be used (expressing whether the state enabling the transition is reachable).

Similarly to the previous example, LoLA cannot reach the state specified by the formula. Given the loop existing between the activation of contexts A and B implies that every time the contexts are active, there is a remaining token in either the temporary place Pr(A) or Pr(B). Thus it is not possible to reach a state in which only A1, B1 and Pr(¬ A) are marked in Figure 7.3.

7.2 Automating Analysis of System Properties

The analyses described in the previous section all take into account the context dependency relations in order to know which kind of transition or state should be used for each analysis input file. However, writing each of the different analysis test cases for a full system can become cumbersome, time consuming and error prone, lessening the advantages of using an analysis tool over a manual verification of the system. Accidental interaction between contexts could be missed in the definition of such files. Thankfully, the formal definition of context dependency relations and their close relation with the system execution enable us to automate the generation of the test cases to be analyzed by LoLA.

7.2.1 Analysis of Context Dependency Relations

In CoPNs, each context dependency relation unequivocally describes a set of properties that should be satisfied whenever contexts are activated and deactivated. These properties can be used to automatically generate the relevant test cases for each of the existing context dependency relations. In the remainder of this section we explain the automatic generation of test cases for the different context dependency relations.

Exclusion dependency relation

Given two contexts A and B and an exclusion dependency relation, $\langle E, \mathcal{C}_A, \mathcal{C}_B \rangle$ between their singleton CoPNs $\mathcal{C}_A$ and $\mathcal{C}_B$. According to the expected behavior of the dependency relation defined by the $\texttt{cons}_E$ function (remember that the $\texttt{ext}_E$ function does not modify the CoPN), the contexts involved in an exclusion dependency relation should not be active at the same time. Six properties must be verified in this case:

1. We need to verify if it is possible to yield a state in which the two contexts could be active at the same time. This is done by means of analyzing reachability of the following marking.

   ```
   ANALYSE MARKING  A1:1, B1:1
   ```

   If the constraints of the exclusion dependency relation are satisfied by the analyzed CoPN then the state should not be reachable.

2. The previous analysis can only provide information about the prohibited states of the system, but not about the permitted states. It is necessary to verify if the accepted cases can occur. This means verifying if one of the contexts could be active while the other is not. This is verified by the following state predicate.

   ```
   FORMULA ( A1 = 1 AND  B0 = 1 ) OR ( B1 = 1 AND A0 = 1 )
   ```

   This formula must always be satisfied in the system. Note that we do not take into account the case in which this the two contexts are inactive, since this is the initial marking of the Petri net and it is satisfied trivially (the analysis wont provide any meaningful results).

3. Additionally, we generate a liveness test case, similar to that of Snippet 7.3, for each of the transitions act(A), act(B), deac(A), and deac(B). In fact, for each of these transitions, the system automatically generates k test cases, one for each of the generated transitions activating or deactivating the context, where k is the capacity given to each context. If the result of the transition analysis is that the transitions are not dead, but the witness step in which the transition can fire is not consistent, then it is possible to analyze the transitions using a reachability analysis.

Causality dependency relation

Given two contexts A and B and a causality dependency relation $\langle C, \mathcal{C}_A, \mathcal{C}_B \rangle$, between their singleton CoPNs $\mathcal{C}_A$ and $\mathcal{C}_B$. According to the expected behavior of the dependency relation defined by the $\texttt{ext}_C$ and $\texttt{cons}_C$ functions, six properties must be verified:

1. Equation (6.8) states that whenever context A is activated, context B is also activated. To analyze this property we use the reachability analysis of the following marking.

   ```
   ANALYSE MARKING  A1:1, B1:1
   ```

   If reachable, the sequence of consistent transition firing must subsume the step $\Upsilon = m_0[req(\texttt{A})act(\texttt{A})0act(\texttt{B})0\rangle$ for some initial marking m_0.

   This test case is complemented by testing that the only way to reach a situation in which context A is active and context B is inactive is by first activating A, and then independently deactivating B.

   ```
   ANALYSE MARKING  A1:1, B0:1
   ```

The sequence of transition firing to reach this state must subsume $\Upsilon = m_0[req(\texttt{A})act(\texttt{A})0act(\texttt{B})0req(\neg\texttt{B})deac(\texttt{B})1\rangle$ for some initial marking m_0.

2. Similar to the first case, Equation (6.7) states that whenever `A` is deactivated, `B` is also deactivated. However, since `B` can be activated and deactivated independently, using a reachability analysis becomes more difficult. It is not clear what the final state of the system representing this property should be. Nonetheless, using a test case similar to Snippet 7.3, it is possible to check if the two right-most deactivation transitions of Figure 6.4 are live or not. k test cases are generated for each transition, where k is the bound of context `B`.

3. The $\texttt{ext}_C$ function dictates that if context `B` is not active and `A` is preparing to deactivate, then `A` can be deactivated without affecting the state of `B`. We use reachability analysis to test if the marking enabling the internal transition deac(`A`) between the two contexts in Figure 6.4 is reachable from an empty initial marking.

```
ANALYSE MARKING  A:1, B0:1, Pr(¬ A):1
```

   If this marking is reachable, the series of consistent transition firings have to contain the steps $\Upsilon_1 = m[req(\texttt{A})act(\texttt{A})act(\texttt{B})0\rangle$, $\Upsilon_2 = m'[req(\neg\texttt{B})deac(\texttt{B})1\rangle$, and the firing of transition $m''[req(\neg\texttt{A})\rangle$, for three reachable markings m, m' and m''. This analysis is complemented by a liveness analysis of the deac(`A`) transition.

4. Additionally, we generate test cases for the liveness analysis of the activation transitions act(`A`) and act(`B`). If context `B` is given a capacity of k, then k test cases are generated for the act(`B`) transition.

Implication dependency relation

Given two contexts `A` and `B` and a implication dependency relation $\langle I, \mathcal{C}_A, \mathcal{C}_B\rangle$, between their singleton CoPN $\mathcal{C}_A$ and $\mathcal{C}_B$. According to the expected behavior of the context dependency relation defined by the $\texttt{ext}_I$ and $\texttt{cons}_I$ functions, seven properties must be verified:

1. Equation (6.10) states that whenever context `A` is activated, context `B` is also activated. To analyze this property we use the reachability analysis of the following marking.

```
ANALYSE MARKING  A:1, B1:1
```

   If reachable, the sequence of consistent transition firing must contain the step $\Upsilon = m_0[req(\texttt{A})act(\texttt{A})act(\texttt{B})0\rangle$, from an initial marking m_0.

2. Equation (6.11) states that whenever `A` is deactivated, `B` is also deactivated. This property is equivalent to the second property of the cause dependency relation. We use a liveness analysis to check if the two right-most deactivation transitions of Figure 6.5 are live or not. If context `B` has a capacity of k, then k test cases are generated for each transition.

3. The deactivation transition of the source context introduced by $\texttt{ext}_I$ dictates that if context B is neither active nor it is preparing to activate, then A cannot not be active. We use reachability analysis to test if the marking enabling the internal transition deac(A) between the two contexts in Figure 6.5 is reachable.

   ```
   ANALYSE MARKING  A:1, B0:1, Pr(B)0:1
   ```

   If such a marking is reachable, the series of transition firings to reach the state must contain the steps $\Upsilon_1 = m[req(\texttt{A})act(\texttt{A})act(\texttt{B})0\rangle$ and $\Upsilon_2 = m'[req(\neg\texttt{B})deac(\texttt{B})1\rangle$ for two reachable markings m and m'. This analysis is complemented by a liveness analysis of the deac(A) transition.

4. Additionally, we generate test cases for the liveness analysis of the activation transitions act(A) and act(B). If B has a capacity of k, then k test cases are generated for the act(B) transition.

Requirement dependency relation

Given two contexts A and B and a requirement dependency relation $\langle Q, \mathcal{C}_A, \mathcal{C}_B\rangle$, between their singleton CoPNs $\mathcal{C}_A$ and $\mathcal{C}_B$. According to the expected behavior of the context dependency relation defined by the $\texttt{ext}_Q$ and $\texttt{cons}_Q$ functions, six properties must be verified:

1. Equations (6.13) and (6.15) state that context A may only be activated if B is already active. To analyze this property we use state predicate analysis to verify if it is possible to reach a state in which B is not marked, and A is.

   ```
   FORMULA  A > 0 AND B0 = 1
   ```

   This formula should not be valid in the CoPN.

2. The transition introduced by the $\texttt{ext}_Q$ function dictates that if context B is not active, then context A cannot not be active. This property is verified by means of the results from the previous analysis and complementing them with the liveness analysis of the deac(A) transition between the two contexts in Figure 6.6.

3. Additionally, we generate test cases for the liveness analysis of the act(A), act(B), deac(B) transitions. For each of these transitions we generate k test cases, where k is the capacity of context B. An additional test case is generated for the deac(A) transition.

Conjunction dependency relation

Given a set of contexts $\{\texttt{A}_1, \ldots, \texttt{A}_n\}$ and a conjunction dependency relation $\langle \wedge, \mathcal{C}_{A_1}, \ldots, \mathcal{C}_{A_n}\rangle$, between their singleton CoPNs $\mathcal{C}_{A_j}$ for $1 \leq j \leq n$. Following the expected behavior of the context dependency relation defined by the $\texttt{ext}_\wedge$ and $\texttt{cons}_\wedge$ functions, five properties must be verified:

1. The context place introduced by the $\texttt{ext}_\wedge$ can only be marked if all of the the contexts places A_j for $1 \leq j \leq n$ are marked. We use state predicate analysis to test whether it is possible to mark the introduced context place when one of the component contexts is inactive.

   `FORMULA` $\texttt{A}_1 \cdots \texttt{A}_n\texttt{1} = \texttt{1}$ `AND (`$\texttt{A}_1 = \texttt{0}$ `OR` $\cdots$ `OR` $\texttt{A}_n = \texttt{0}$`)`

   Such an state cannot be reached in the CoPN representing a conjunction dependency relation of different contexts.

2. Similarly to the previous case, we would like to test if it is possible that all component contexts are active, but the conjunction context is not. This property also uses a state predicate analysis.

   `FORMULA` $\texttt{A}_1 \cdots \texttt{A}_n\texttt{1} = \texttt{0}$ `AND (`$\texttt{A}_1 > \texttt{0}$ `AND` $\cdots$ `AND` $\texttt{A}_n > \texttt{0}$`)`

   As in the previous test case, such an state cannot be reached in the CoPN representing a conjunction dependency relation of different contexts.

3. Liveness test cases are generated for the transitions $\text{act}(A_1 \cdots A_n)$ and $\text{deac}(A_1 \cdots A_n)$ introduced by the $\texttt{ext}_\wedge$ function. The activation and deactivation transitions of all the component contexts are also verified using liveness analysis.

Test case generation for composed CoPNs

Generation of more general test cases for any CoPN is a more challenging task. The interactions between adaptations can be given by multiple combinations of context dependency relations among them, and hence the generation of the test cases must take into account all these combinations. For this reason, in its current state, CoPNs are only able to generate test cases for the properties of the defined context dependency relations in the system, and sets of **transitive context dependency relations**.

Definition 7.3. *Given three contexts* `A`, `B` *and* `C` *with respective singleton CoPNs* $\mathcal{C}_A, \mathcal{C}_B$, *and* $\mathcal{C}_C$, *and a context dependency relation* R *such that* $\langle R, \mathcal{C}_A, \mathcal{C}_B \rangle$ *and* $\langle R, \mathcal{C}_B, \mathcal{C}_C \rangle$. R *is said to be transitive if and only if the activation/deactivation of* `A` *influences the activation/deactivation of* `C`.

Example 7.9. An example of a transitive context dependency relation as described in Definition 7.3, is the requirement dependency relation. Let us take three contexts `A`, `B` and `C` such that `B`–◀`A` and `C`–◀`B`.

The first requirement dependency relation states that in order to activate `B`, `A` must be active first; and that if `A` is inactive, `B` also is. We use this information to analyze the second requirement dependency relation. In the second relation in order to activate `C`, `B` must be active first, which can only happen if `A` is already active. In the normal case, context `C` is ensured to be inactive if `B` is inactive, which at first hand does not relate to `A`. However, since `A` being inactive ensures that `B` is inactive, it is possible to conclude that every time `A` is inactive then `C` is inactive.

Example 7.10. The exclusion dependency relation is not transitive. For three contexts `A`, `B` and `C` such that `A`□–□`B` and `B`□–□`C`, the activation state of `A` and `C` are independent. That is, activation or deactivation of one context does not affect the activation state of the other. In particular, `A` and `C` can be active at the same time.

In CoPN three context dependency relations are transitive, namely, implication, causality, and requirement. Using the transitive property for context dependency relations we are able to generate test cases for more general CoPNs as follows:

1. For an implication dependency relation, we use reachability analysis to verify the correct activation of all contexts in its transitive closure —that is, all contexts reachable via a chain of implication dependency relations. This analysis requires to take a "root" context as a starting point, and to take into account all target contexts related to it by means of an implication dependency relation: all target contexts related to those contexts via an implication dependency relation, and so on. As a way to better visualize which contexts are part of the analysis test case, imagine a tree structure where edges are implication dependency relations and the root of the tree is a context which is not a target of any implication dependency relation, as shown in Figure 7.4.

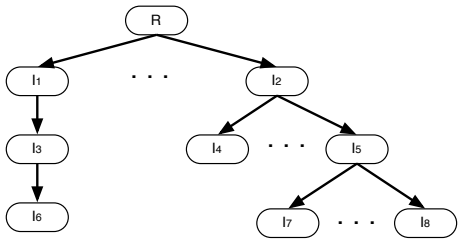


Figure 7.4: Tree of implication dependency relations.

For the analysis we generate a state predicate of the form:

`FORMULA R > 0 AND` I_1 `> 0 AND ... AND` I_n `> 0`

Where `R` is the root node of the tree and the I_j for $1 \leq j \leq n$ are all the contexts in the transitive closure of the implication dependency relation from the root node —that is, an enumeration of all contexts reachable from the root context in Figure 7.4. A test case as the one presented here, is generated for each root node of an implication dependency relation in the composed CoPN.

Generating a test case for the deactivation of the contexts is more challenging. As it occurred for the generation of the test cases for an implication dependency relation, there is no state that unequivocally identifies the deactivation of all contexts. A liveness analysis is required to verify (individually) each deactivation transition. However, it is possible to define a general analysis for the inactive states of target contexts. For each branch to the root node in Figure 7.4 we generate a state formula of the form:

$$\texttt{FORMULA}\quad \mathtt{I}_l = 0 \ \texttt{AND}\ Pr(\mathtt{I}_l) = 0 \ \texttt{AND}\ (\mathtt{I}_1 > 0 \ \texttt{OR}\ \ldots\ \texttt{OR}\ \mathtt{I}_n > 0)$$

In this formula $\mathtt{I}_l$ represents a leaf of the tree and the $\mathtt{I}_1, \ldots, \mathtt{I}_n$ nodes represent an enumeration of the branch of the tree branch leading from the root node to the leaf I_l. The formula verifies that it is not possible for a context in a branch to be active, if the tree leaf for that branch is inactive. The combination of all generated test cases verifies that a context must be inactive if any of the leafs reachable from it are inactive.

2. Similar analyses as those generated for the implication dependency relation can be generated for the causality dependency relation. Using a transitive closure for causality dependency relations, similar to the one shown in Figure 7.4, it is possible generate the state predicate formula for a reachability analysis of context activations.

$$\texttt{FORMULA}\quad \mathtt{R} > 0 \ \texttt{AND}\ \mathtt{I}_1 > 0 \ \texttt{AND}\ \ldots\ \texttt{AND}\ \mathtt{I}_n > 0$$

This formula states that if the context root to a set of contexts in a causality dependency relation is activated, then all such contexts must be activated. In the case of causality dependency relations, however, it is not possible to generate a general analysis test case for the verification of context deactivations because target contexts could have a greater activation count than the source contexts (LoLA does not allow us to create formulae comparing the states of places).

3. Reachability analysis is used to verify the transitive closure of requirement dependency relations, a tree similar to Figure 7.4 where every edge is a requirement dependency relation, where a top level node is required by a lower level node. To verify the activation state of the contexts we generate a state predicate test case for each branch of the tree.

$$\texttt{FORMULA}\quad \mathtt{R} = 0 \ \texttt{AND}\ \mathtt{I}_1 > 0 \ \texttt{AND}\ \ldots\ \texttt{AND}\ \mathtt{I}_n > 0$$

This formula verifies that every context in the branch $\mathtt{I}_1$ to $\mathtt{I}_n$ can only be active if the preceding context, and ultimately the root context, are active. This formula also represents the state triggering the deactivation of the required contexts. A liveness test of the deactivation transition for each of the contexts I_j for $1 \leq j \leq n$, provides an insight of the validity of the state. If the transition is live, then the state expressed by this state predicate formula is a valid state of the system. If the transition is dead, the state expressed by the state predicate formula is not reachable.

7.2.2 Analyses Integration

In order to perform the reachability and liveness analyses described in Section 7.1.2 we use the external reasoning tool, LoLA, which provides comprehensive support for such Petri net analyses.

To enable the analysis of system properties, our CoPN tool suit can automatically generate the (basic) test cases, and perform the unfolding of the CoPN to a bounded Petri net without inhibitor arcs and stripped down of its reactive and priority semantics. These are used as input files for the external reasoning tool LoLA. The execution of the analyses is, however, not automatic. Although it would be possible to automatically connect the CoPN model with LoLA, we decided not to do it for the moment. Next we discus our decision.

This chapter envisions a process to reason about system properties as additional support to the run-time consistency management described in Chapter 6. We use an early reasoning approach to support the conception and design of the development processes of COP systems. The analyses presented here are meant for the identification of incoherences while modeling dynamic adaptations and the interactions among them. Using a fully automated process would improve the analysis process of the system —that is, reduce the burden on manually checking each test case. Furthermore, the CoPN model could be extended to provide the means to suggest or automatically resolve incoherent configurations of contexts, allowing the introduction of new contexts at run time. An automatic process could allow us to modify the structure of the CoPN and ensure that the remaining contexts composed in the system continue to be coherent.

However, LoLA allows the analysis of multiple Petri net properties in the case of basic Place/Transition nets. Direct integration between the two systems would not be very fruitful because the available analyses do not completely cover the semantics of CoPNs. Some properties require a careful analysis with respect to the CoPN semantics. More specialized techniques than those provided by LoLA are necessary to support analyses with the specific semantics of CoPN. Although specialized analysis techniques could be provided to better capture the (reduced) semantics of CoPN, the design and development of Petri net analyses techniques fall outside the objectives of our thesis, hence, we decided not to integrate them.

Furthermore, CoPN is only able to generate the basic test cases for the analysis of the system properties. The test cases are based on the constraints formally defined in CoPNs. However, the analysis of complete systems may require more complex test cases that cannot be deduced directly from the CoPN structure and formalization, but requires domain knowledge on the part of the programmer. An automatic analysis process would still require a way to generate a more comprehensive set of analysis test cases, or integrate those test cases defined by programmers.

7.3 Conclusion

This chapter developped the necessary concepts to enable the partial analysis of COP systems. The different analyses presented in this chapter are envisioned as a complementary support to the run-time consistency management described in Chapter 6. The purpose of an early analysis of CoPN properties is to identify incorrect definitions of interactions between contexts and incoherent compositions of adaptations during the design of the system. In particular it enables us to identify (1) situations in which adaptations are not able to take place because of the constraints imposed on their activation are never satisfied, and (2) situations in which context activations lead to a infinite sequence of transition firings, blocking or crashing the system. These situations are of particular interest because they cause inconsistencies in the predictability of the system and are hard to identify or avoid during the execution of CoPNs.

CoPN takes advantage of existing tools for the analysis of Petri nets, for the analysis of system properties. The analysis of COP systems does not need to be developed from scratch, but builds on existing techniques. By means of net unfolding and the use of the Low Level Petri net Analyzer, it is possible to reason about a restricted semantics of CoPNs and their properties. This reasoning can be used to provide insights about the interaction of behavioral adaptations in a COP system. meeting the safety Requirement (**M.2**) defined in Section 2.4.

This chapter presented two analysis techniques of Petri nets that are effectively used to identify the activation states of contexts and the possibility to execute particular actions over contexts. We enable the automatic generation of test cases based on the formal definition of the context dependency relations composing the system. In case an incoherence is identified while analyzing one of said test cases, developers can track the incoherence back to a particular context and its context dependency relations, allowing them to modify the interaction of the contexts in order to solve identified incoherences. Through reachability analysis it is possible to reason about expected or unexpected states of the contexts composed in a CoPN. Through liveness analysis it is possible to reason about the availability of particular actions over a context in a CoPN. In the current incarnation of the CoPN model, the analysis of these properties is delegated to LoLA, an external Petri net analyzer. The analyses provided for the CoPN model comply with the decision Requirement (**M.4**) defined in Section 2.4.

However useful the analyses of CoPNs has proven to be, the fact that the semantics of our model has to be restricted only allows us to have an approximation of the analyzed properties. To solve this problem it could be interesting to use reset arcs [56] instead of inhibitor arcs, and perform our analyses in the Petri net with reset arcs or Reset nets. The main function of reset arcs between a transition and a place is to empty the place whenever the transition fires. It has been proven that the coverability problem is decidable in Reset nets. The analysis of Reset nets provides an over-approximation of the cover-

ability problem, that could be used in the case of CoPN. The use of reset arcs should be studied further in CoPN, a discussion on their usefulness can be found in Section 11.3.2.

The reasoning engine presented in this chapter provides support for the *safety* and *decision* requirements for run-time models for Dynamically Adaptive Software Systems, increasing their predictability and confidence. The analyses used in CoPNs allow us to identify incoherences that could not be otherwise identified at run time.

Chapter 8

A Comprehensive Programming Model for Dynamically Adaptive Software Systems

Chapters 6 and 7 introduced the run-time management of context activations and the design-time identification of inconsistencies, of the CoPN programming model. In this chapter we explore the capabilities of CoPNs as a programming model for Dynamically Adaptive Software Systems. This chapter overviews CoPNs from two perspectives. First, it provides the details of CoPNs in the light of the architecture and adaptation process for Dynamically Adaptive Software Systems, presenting CoPNs as a comprehensive programming model for such systems, and in particular COP. Second, it gives an overview on the support provided for the development of Dynamically Adaptive Software Systems.

This chapter revisits the adaptation process and architecture for developing Dynamically Adaptive Software Systems [159, 161], and its refinements proposed in several COP systems [74, 27, 77], normally referred to as the context-awareness architecture. CoPNs are mapped to this architecture as a means to demonstrate their appropriateness as a comprehensive model for the development of Dynamically Adaptive Software Systems. We explain the way in which CoPNs comply with each of the components of context-awareness architecture by giving the details of the CoPN's model implementation.

The second part of this chapter explores the support provided by the model as part of the software development process. In particular we explore the capabilities of CoPNs to ease testing Dynamically Adaptive Software Systems, and more precisely the dynamic activation of adaptations. The exploration of testing the dynamics of context activations follows from its proximity with the overall concern of this dissertation —managing behavioral inconsistencies. Here, we present a simulation tool that allows to manually activate and deactivate contexts and observe their interactions.

8.1 Architecture for Dynamic Adaptations Revisited

In the development of the formal basis presented in CoPN, we also envisioned a single programming model that comprehends the totality of COP systems. This is motivated by the observation that current context-aware systems focus either on the dynamic adaptation of behavior, or on the definition and propagation of changing situations in the surrounding execution environment of the system. We argue that in order to provide a comprehensive model for the development of Dynamically Adaptive Software Systems both aspects are required. This section explores the programming model of CoPNs, presenting the implementation aspects of the formal basis presented in Chapters 6 and 7, and exploring the way in which CoPNs enfold other aspects of the of the adaptation process, such as discovery and gathering of context information. Figure 8.1 shows the general architecture of CoPNs.

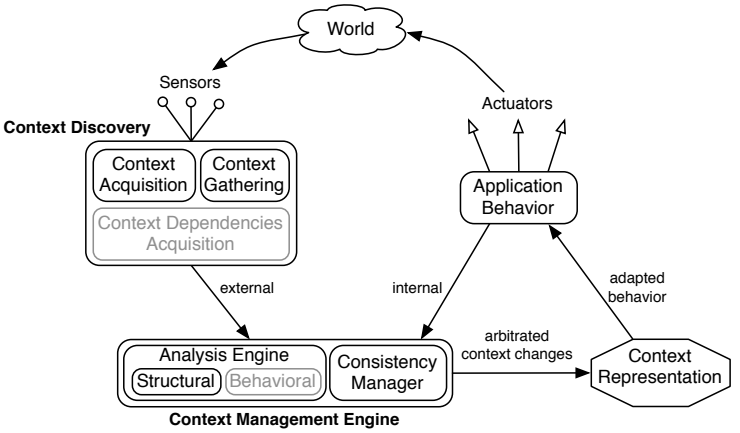


Figure 8.1: Revisited architecture of context-awareness with CoPNs.

CoPNs use the same reference architecture for Dynamically Adaptive Software Systems presented in Section 4.4.1 [74, 27, 77]. Nonetheless, in Figure 8.1 we extend the original proposed architecture with the specification of required modules to fulfill the tasks of the architecture components: *context discovery*, *context management engine*, *context representation*, and *application behavior*. An in-depth explanation of how each component is realized in CoPNs, with its internal modules, is given in the remainder of this section.

8.1.1 CoPN Application Behavior

The application behavior component of the context-awareness architecture proposed by the CoPN model relies on the process already existing in Subjective-C, as explained in Section 4.4.1. The observable behavior of Dynamically Adaptive Software Systems corresponds to the composition of the behavioral adaptations

associated to *all active* situations gathered from the surrounding execution environment of the system. From the perspective of end-users, the application behavior component provided by CoPNs presents an intangible difference with respect to the application behavior component provided by Subjective-C. Nonetheless, the CoPN model makes a contribution in ensuring that the application behavior component is consistent according to the defined context dependency relations and the situations gathered from the surrounding execution environment of the system. This contribution differentiates the CoPN model from traditional models in that the observed application behavior in the CoPN is not that of all situations present in the surrounding execution environment of the system, but rather is composed of *all the situations* from the surrounding execution environment *that lead to a consistent state* of the system, as expressed in Definition 6.20.

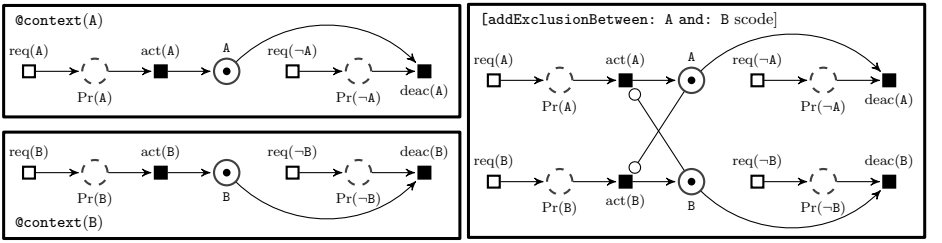
8.1.2 CoPN Context Representation

As mentioned in Section 4.4.1, the context representation component of the context-awareness architecture keeps track of the active contexts and their associated behavioral adaptations. That is, it keeps track of the woven behavioral adaptations in the system at any point during its execution. The CoPN model follows the same approach for the representation of contexts as other COP languages such as Ambience [74] or ContextErlang [163].

CoPNs present two representations of the adaptations defined in the system. The first representation of adaptations comprehends all contexts defined in the system. Such representation is given by the underlying Petri net structure itself. Every time a context or a context dependency relation is defined in the system, the structural representation of contexts is updated to include the new context dependency relation definition. Context definitions, such as `@context(A)`, are represented by an individual CoPN. Figure 8.2a shows the definition of two CoPNs, where each independent CoPN is delimited by a rounded square. Defining context dependency relations between contexts, for example, an exclusion dependency relation `A□–□B`, generates a CoPN consisting of two contexts and the connections between them, as shown in Figure 8.2b. The generation of a CoPN through the definition of context dependency relations is related to the high-level definition of context dependency graphs of Subjective-C. However, the representation given by the CoPN excels the representation of context dependency graphs by encoding the semantics of the defined context dependency relations composed into the CoPN.

The second representation of adaptations provided by CoPNs concerns those adaptations that are currently active in the system. To this end we take advantage of the Petri net marking, which provides a live representation of the system state. The set of active adaptations corresponds to the set of marked context places. The set of active adaptations is kept as part of the state of the system in CoPNs, this corresponds to the stable marking, given in Definition 6.22.

The representation of adaptations provided by the CoPN model is an im-



(a) Two context definitions represented as singleton CoPNs.

(b) An exclusion dependency relation represented as a CoPN.

Figure 8.2: Structural representation of contexts and context dependency relations definition in CoPN.

provement over the context representations proposed by previous COP approaches [74, 98, 163]. First, context representations, usually, require the introduction of two data structures, one data structure represents the adaptations and their relations, and another data structure keeps track of which adaptations are active. CoPNs use a representation of active contexts that is already embedded with the structural representation of adaptations, both context representations are derived from the same data structure. Second, the context representation through the Petri net marking, does not only visualize the active contexts of the system, but it also provides a first hand view on the activation count of each context —a technique used to keep track of context activations in the light of context interaction [33]. Third, CoPNs allow the possibility of recreating activations and deactivations of contexts by keeping track of the consistent steps generated during the execution of the system. Such information, currently not provided by other COP systems, could be used for simulation and testing purposes with respect to the activation of contexts, or to cache activations and deactivations in order to ease their verification.

The context representation module constitutes the core of the CoPN programming model. As said earlier, the context representation consists of the Petri net itself. In CoPNs, contexts (and their representation), are given by an ensemble of places and transitions, as shown in Figure 8.3. The basic structure of the CoPN implementation is inspired by the ideas of the ePNK (Petri Net Kernel) [112] and SNAKES [147]. CoPNs are implemented as incidence vectors for the representation of the Petri net graph, where each transition is aware of its neighboring places, and each place is aware of its marking set.

A context, `SCContext` is represented as a set of three places and four transitions, as given in Definition 6.2. Additionally, to comply with the definition of CoPNs given in Section 6.1, we introduce a specialization of places as context places (`PNContextPlace`) and temporary places (`PNTemporaryPlace`), and transitions as external transitions (`PNExternalTransitions`) and internal transitions (`PNInternalTransition`). Note that external and internal transitions are used to provide the reactive semantics of CoPNs, but different transition pri-

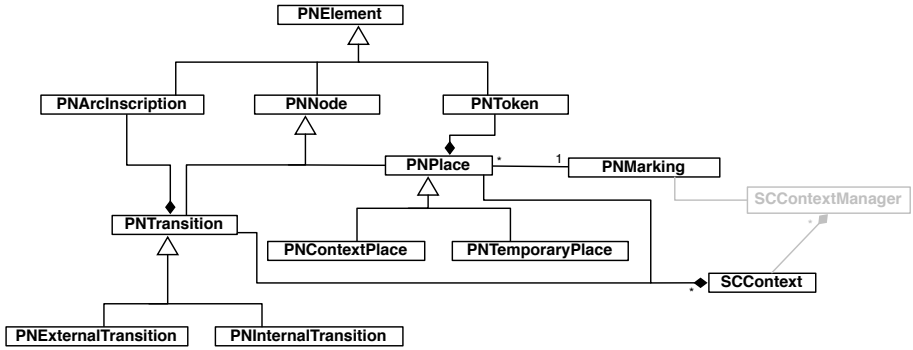


Figure 8.3: Structure diagram of the context representation component in CoPN.

orities can be given to each transition type. However, an internal and external transition can never have the same priority.

8.1.3 CoPN Context Management Engine

The context management engine component of the context-awareness architecture is presented as combination between an internal and an external engine of the system [159]. It is an internal engine as it provides language features for the definition of adaptations. However, the logic defining and managing the contexts and behavioral adaptations is independent from the rest of the system's logic, making it an external engine. So far, the concrete proposals for a management entity in COP languages have been restricted to the definition of context changes throughout specified sets of rules, similar to those for context dependency relations, described in Section 4.4.1. However, we observe that current approaches for the management of safe context configurations rely on definitions provided by programmers to be correct [163]. Even if it is possible to verify some properties the system must satisfy, such properties are specified by programmers as part of an independent system [49, 103]. As described in Requirements **M.3** and **M.4**, we envision a programming model that allows to abstract and reason upon the system in a comprehensive way. From these requirements we conclude that the management of context activations provided by other COP approaches is not sufficient. Section 4.2 motivates this fact by describing situations in which the rules defined by programmers may lead to inconsistencies of the system's behavior. We argue that the management of behavioral adaptations must be based on a formal definition of rules describing context interactions, and their automated verification.

Similar to the proposal of Ambience [74], CoPNs enclose the consistency management module into a broader component which comprehends the complete logic for realizing dynamic adaptations in the system. As shown in Figure 8.1, the context management engine consists of two modules. The analysis engine and the consistency manager.

The motivation behind explicitly providing two modules for the management of Dynamically Adaptive Software Systems is raised by two different concerns with respect to the adaptation logic of the system. On the one hand we want to manage the inclusion and withdrawal of adaptations dynamically at run time. Moreover we want to arbitrate such processes to ensure the consistency of the system's behavior. On the other hand we want to reason about the adaptations defined in the system, and the interactions between them. It is natural to separate the two concerns into two different modules, one module for the management concern, and another module for the analysis concern. Since the management of adaptations activations takes place at run time, the separation of the system in two modules allows us to defer part of the costly analysis tasks to earlier stages of the development cycle, such as design or compile time. Analysis of dynamic systems in two (or more) phases (design time and run time) has already been proven a successful technique applied in similar approaches [22, 64].

The context manager module is responsible for arbitrating the system at run time, as described in Chapter 6. This module is the central entity of the CoPN model. The responsibilities of the context manager (`SCContextManager`) comprise the definition of context and context dependency relations, and the management of the activation and deactivation of contexts. To do this, the context manager holds a reference to the representation of all contexts (`SCContext`) and the set of active contexts (`PNMarking`). The structure of the context manager module is shown in the left-hand side of Figure 8.4. We continue by explaining each of the responsibilities of the context manager module in detail.

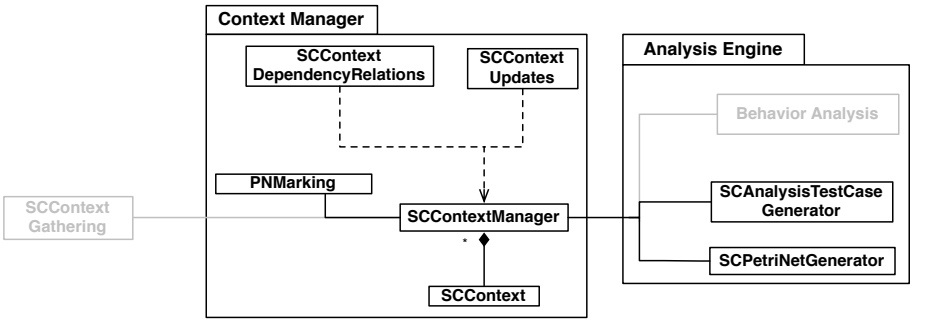


Figure 8.4: Implementation of the context management component in CoPN.

First of all, definition of contexts is done by creating the structure of a singleton CoPN (Definition 6.2), as for example, the one shown for context `A` in Figure 8.2a. Snippet 8.1 shows the definition each context object (`SCContext`), given by its three places (Lines 5 through 7) and four transition (Lines 9 through 12).

Secondly, definition of context dependency relations is not hierarchical as is the case in Subjective-C [123]. In Subjective-C context dependency relations are explicitly objects in the class hierarchy of the context manager. CoPNs flatten

```
1 - (void)addContextWithName:(NSString *) contextName andBound:int bound {
2   self.contextName = contextName;
3   self.bound = bound;
4   //create places
5   PNContextPlace *c = [PNContextPlace initWithName:contextName];
6   PNTemporaryPlace *pr = [PNTemporaryPlace initWithName:[self getPrName:↩
        contextName]];
7   PNTemporaryPlace *prn = [PNTemporaryPlace initWithName:[self getPrnName↩
        :contextName]];
8   //create transitions
9   PNTransition *req = [PNExternalTransition initWithName:[self getReqName↩
        :contextName]];
10  PNTransition *reqn = [PNExternalTransition initWithName:[self ↩
        getReqnName:contextName]];
11  PNTransition *act = [PNInternalTransition initWithName:[self getActName↩
        :contextName]];
12  PNTransition *deac = [PNInternalTransition initWithName:[self ↩
        getDeacName:contextName]];
13  // connecting places and transitions
14  [req addOutput:NORMAL_ARC toPlace:pr];
15  [act addInput:NORMAL_ARC fromPlace:pr];
16  [act addOutput:NORMAL_ARC toPlace:aContext];
17  [reqn addOutput:NORMAL_ARC toPlace:prn];
18  [deac addInput:NORMAL_ARC fromPlace:aContext];
19  [deac addInput:NORMAL_ARC fromPlace:prn];
20 }
```

Snippet 8.1: Declaration of a context as a singleton CoPN.

context dependency relations (`SCContextDependencyRelations`) as function applications over a set of singleton CoPN, as described in Section 6.2. Context dependency relations are described declaratively defining the structure of the CoPN that satisfies the intended interaction between its contexts. As an example, Snippet 8.2 shows the declaration of an implication dependency relation. Lines 2 through 12 define the extension function $\texttt{ext}_I$, and Lines 14 through 27 define the constraining function $\texttt{cons}_I$ for the implication dependency relation.

In Snippet 8.2, Lines 3 and 15 ensure the restriction that a context cannot have a context dependency relation with itself. The constraints that must be satisfied by the CoPN are verified programmatically over all elements of the CoPN —that is, over all places and transitions after the application of the $\texttt{union}(S)$ and $\texttt{ext}(\texttt{union}(S), \mathcal{R})$, for given sets of singleton CoPNs, S, and context dependency relations, $\mathcal{R}$, as respectively given in Definitions 6.10 and 6.11. In the case of an implication dependency relation, for example, Lines 18 through 21 verify that all activations of the source context request the activation of the target context. All transitions of the CoPN which have the context place of the source context as an output, and for which this place is not an input, are given an arc from the transition to the preparing to activate temporary place of the target context.

Thirdly, context activations are received from the context discovery via the context gathering module, which is explained later in this section. Instead of having specific processes for the activation and deactivation of contexts, the CoPN programming model unifies the two processes as context updates (`SCContextUpdates`).

All changes in the surrounding execution environment of the system are al-

```
1  //Implication ext associated function
2  - (void) addImplicationFrom:(SCContext *)source to:(SCContext *)target {
3    if([source isEqual:target]) return;
4
5    //add additional transition deac(source)
6    PNInternalTransition *deacSource = [PNInternalTransition initWithName: ↩
         deac];
7    //add arc to flow function f
8    [deacSource addInput:NORMAL_ARC fromPlace: source];
9    //add arcs to inhibitor flow function f_o
10   [deacSource addInput:INHIBITOR_ARC fromPlace:target];
11   [deacSource addInput:INHIBITOR_ARC fromPlace:[target ↩
         getPrepareForActivation]];
12 }
13 //Implication cons associated function
14 - (void) implicationConstraintsFrom:(SCContext *)source to:(SCContext *)↩
       target {
15   if([source isEqual:target]) return;
16
17   //add arcs to activate the target when the source is activated
18   for(PNTransition *t in [self getInputsForPlace:source]) {
19     if(![[t inputs] containsObject: source])
20       [t addOutput:NORMAL_ARC toPlace:[target getPrepareForActivation]];
21   }
22   //add arcs to deactivate the target when the source is deactivated
23   for (PNTransition *t in [self getOutputsForPlace:source]) {
24     if(![[t inhibitorInputs] containsObject:target])
25      [t addOutput:NORMAL_ARC toPlace:[target getPrepareForDeactivation]];
26   }
27 }
```

Snippet 8.2: Declaration of the implication dependency relation's extension and constraining functions.

ways resolved by triggering of an external transition in CoPNs, as described in Section 6.3.1. Snippet 8.3 shows the implementation for a context update —that is, activation or deactivation of a context. Lines 6 through 11 provide the reactive semantics of CoPNs by firing all internal transitions that become enabled. Once there are no more transitions enabled for firing Lines 14 through 24 verify the marking of the CoPN for inconsistencies and rollback all modifications if an inconsistency is found. The set of enabled internal transitions is maintained up to date in Lines 34 through 43.

Three things are worth mentioning from Snippet 8.3. First, CoPNs provide a unified technique for processing events coming from the surrounding execution environment of the system, unlike Subjective-C [123] which provides two independent techniques, one for activation events, as shown in Snippet 4.10, and one (analogous method) used for deactivation events. Second, the activation process of CoPNs does not require to modify manually the activation counter of a context, this is implicitly done by the number of tokens in a place. Third, application of *all* rules for *all* context dependency relations is managed automatically in Snippet 8.3. This is in contrast of other COP languages as Subjective-C [77] or ContextErlang [163], where the interaction between adaptations is encoded in the activation logic. Introduction of new context dependency relations in this approaches requires to modify the activation semantics to account for the new relation.

```
- (BOOL) contextAction: (PNTransition *) transition {
  [transition fire];  //fire external transition -firing rule (6.1)
  [self enabledActions];

  //fire all enabled internal transitions -firing rule (6.2)
  while([transitionSet count] != 0) {
      PNTransition *t = [transitionSet anyObject];
      [t fire];
      [transitionSet removeObject: t];
      [self enabledActions];
  }
  [self updateActiveContexts];

  if([self isStable]) {
      //check for consistency of the step -firing rule (6.4)
      [currentContexts updateSystemState];
      return YES;
  } else {
      //handle error and alert user -firing rule (6.3)
      [currentContexts revertOperation];
      ...
      return NO;
  }
}

- (void) updateActiveContexts {
  [currentContexts clean];
  for(SCContext *c in places) {
      if([[c tokens] count] > 0)
          [currentContexts addActiveContextToMarking:c];
  }
}

- (NSMutableArray *) enabledActions {
  NSMutableArray *result = [[NSMutableArray alloc] init];
  for(PNTransition *transition in transitions) {
      if([transition checkEnabled])
          [result addObject:transition];
      else
          [transitionSet removeObject:transition];
  }
  return [result autorelease];
}
```

Snippet 8.3: CoPN implementation of context updates.

Finally, the context manager module (`SCContextManager`) is responsible for the inclusion and withdrawal of behavioral adaptations in the system. CoPNs rely on the selection of behavioral adaptations used in Subjective-C based on the definition of Subjective-C's methods (Snippet 4.9) which are associated with context objects (`SCContext`). The existing method was adopted in CoPNs because it is tightly related with the context activation component, which the CoPN model also inherited from Subjective-C.

Chapter 7 presents the details of the analysis module introduced by CoPNs. Currently, the analysis of the system properties is delegated to the LoLA analyzer. The analysis module of CoPNs, shown in the right-hand side of Figure 8.4, manages the unfolding of the model into a Petri net with place capacities and without inhibitor arcs (`SCPetriNetGenerator`) and the generation of the test cases to be analyzed by the system (`SCAnalysisTestCaseGenerator`). The Analysis engine interacts with the context manager module to retrieve the infor-

mation about contexts and context dependency relations defined in the system, in order to derive the appropriate analysis test cases. Analysis of properties about the system's adaptations in CoPNs does not require an independent definition of rules to be verified, from those describing interactions between adaptations, as is required in approaches taken by contextL [49] and EventCJ [103]. Currently CoPNs only allow to reason upon structural properties of the system. Section 11.4.2 describes avenues of future work for reasoning about behavioral adaptations defined in the system. The behavioral analysis module is grayed out in Figure 8.4 for this reason.

8.1.4 CoPN Context Discovery

The context discovery component of the context-awareness architecture is in charge of managing all interactions between the system and its surrounding execution environment. Discovery of contexts is divided into four main modules in CoPNs. (1) A context gathering module, which is an active monitor of the information available about the surrounding execution environment of the system. This module generates the context updates according to the information gathered from the surrounding execution environment. (2) A context monitoring module, which is the bridge to communicate information obtained from the sensor network of the system to the context gathering module. (3) A context acquisition module, which is in charge of discovering new contexts and associating their definition with the running CoPN. (4) A context dependency relations acquisition module, which is in charge of discovering *new* definitions of interactions between contexts and associate them with the running CoPN. Figure 8.5 shows the implementation of the context discovery component in CoPNs. In the current version of CoPNs we only covered the context gathering module (`SCContextGathering`) and the context acquisition module (`SCContextAcquisition`). Monitoring information about the surrounding execution environment and the acquisition of context dependency relations, shown in gray in Figure 8.5, are not yet fully supported, and are thus left out of our discussion below.

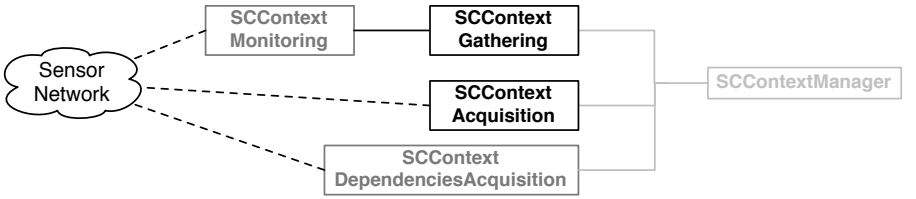


Figure 8.5: Implementation of the context discovery component in CoPN.

Context acquisition module

The context acquisition module is in charge of composing new context definitions, discovered and acquired from the surrounding execution environment of the system with its current CoPN representation. We take advantage of Bonjour,[1] the communication protocol provided by Objective-C, to enable discovery and communication between devices. This communication model is also reused for gathering context information about the surrounding execution environment of the system.

Currently the context acquisition module only allows to send context definitions (e.g., `@context(A)`) and context dependency relations between contexts (e.g., `[addRequirementTo:A of:B]`) from an external source. The context acquisition module addresses Requirement **D.5**. Such definitions are received by the context manager module and integrated with the current CoPN following the composition operation described in Section 6.2.

Example 8.1. As an example let us take the maps application defined in Snippet 6.1. Let S={POSITIONING, NLBS, GPSANTENNA, GSMLOCATION, BLUETOOTH, WLAN, CONNECTIVITY} be the set of singleton CoPNs defined in the application, and $\mathcal{R}$={$<C$, WLAN, CONNECTIVITY>, $<C$, BLUETOOTH, CONNECTIVITY>, $<I$, GSMLOCATION, POSITIONING>, $<I$, GPSANTENNA, POSITIONING>, $<I$, NLBS, POSITIONING>, $<Q$, NLBS, CONNECTIVITY>} be the set of context dependency relations defined between such contexts. The CoPN defined for the maps application is given by $\mathcal{P} = \circ(S, \mathcal{R})$. Suppose that $\mathcal{P}$ is defined in a device $\texttt{dev}_2$. Suppose further that another device, $\texttt{dev}_1$ (e.g., a sensor), is discovered by $\texttt{dev}_2$, where the discovered device provides the definition of a new context, PRIVATE, and an exclusion dependency relation between the PRIVATE and POSITIONING contexts.

As soon as $\texttt{dev}_1$ is discovered it sends the messages `@context(PRIVATE)`, and `[addExclusionBetween:PRIVATE and:POSITIONING]` to $\texttt{dev}_2$. As a result, $\texttt{dev}_2$ composes this context definition with the existing CoPN $\mathcal{P}$, yielding a new CoPN $\mathcal{P}' = \circ(S \cup \{\text{PRIVATE}\}, \mathcal{R} \cup \{< E, \text{PRIVATE}, \text{POSITIONING} >\})$. The visual representation of $\mathcal{P}'$ corresponds to the CoPN shown in Figure 6.11.

Once contexts and context dependency relations have been composed in the running system, these are automatically taken into account for the run-time verification of consistent activations. However, the process of context acquisition module is still a proof-of-concept, and the discovery and definition of new contexts and context dependency relations between existing contexts could be improved. We still assume that all information about contexts and behavioral adaptations is centralized in one context manager module. Even more, definition of new contexts are behavior-less —that is, new context definitions are not created with their associated behavior, and hence, no behavioral adaptations are composed into the system when these contexts become active. This is a

[1] `https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/NetServices/Introduction.html`

limitation of the underlying technique for defining behavioral adaptations used in Subjective-C, which requires all behavioral adaptations to be known at compile time. We discuss how this shortcoming could be addressed using CoPNs in Section 11.3.

Context gathering module

The context gathering module is in charge of retrieving information about the surrounding execution environment of the system (e.g., through a sensor network), and to generate the corresponding context changes based on such information. Each context in the context gathering module is associated with a particular (set) of information about the surrounding execution environment of the system. Association of a context with information in the surrounding execution environment is delegated to the context monitoring module and is currently not supported in CoPNs. Currently, CoPNs simulated gathered context information using an external device that signals context activations and deactivations. Context activations are signaled with the context manager every time the information associated to the context changes. The general process for a context activation is shown in Figure 8.6. An analogous process takes place for deactivating a context.

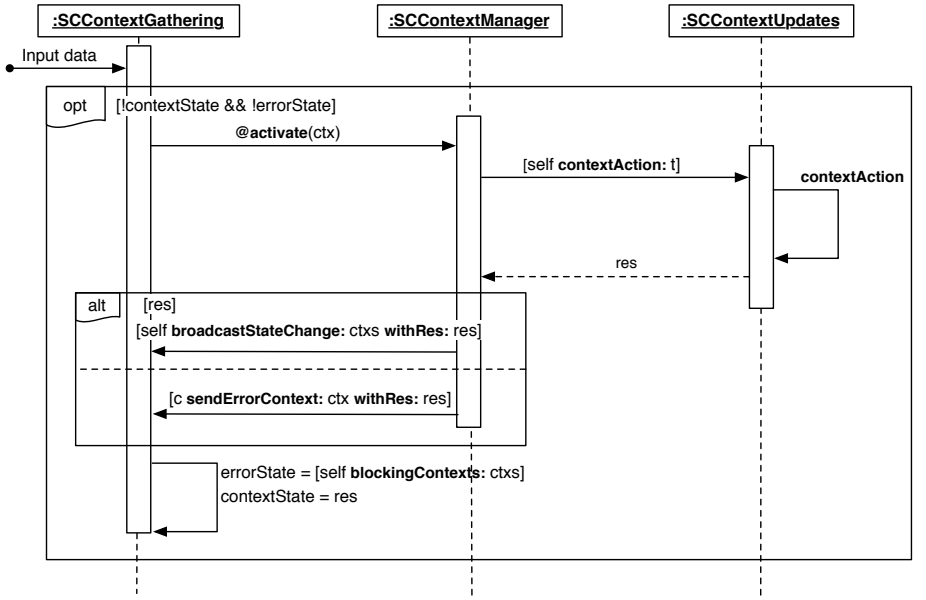


Figure 8.6: Process for context activation.

Each context `ctx` instantiated in the context gathering module has two variables: a `contextState` variable to keep track of the state of the context and an `errorState` variable to signal if the state change did not take place because

a context dependency relation prohibit the context to activate. The purpose of these two variables is to manage the activation/deactivation messages sent to the context manager module, so that the context manager is not overflowed with processing requests from the same context activations. Additionally, using these variables we can ensure that context activation messages are sent to the context manager modules if and only if there is a state change in the surrounding execution environment of the system and it is not already known that the context update is going to be denied. As explained previously in the context management module, there are two possible outcomes from a context activation. The first possibility is that the context activation is a consistent step. In such a case the return variable `res` is set to true, and the new set of active contexts is broadcasted to all contexts defined in the context gathering module. Using this information, each context checks if the state of the contexts impeding its activation or deactivation has changed, in which case the `errorState` variable is changed. The other possibility is that the context activation is not a consistent step. In such case the `res` variable is set to false, and the name of the context impeding the activation or deactivation is sent to the context entity in the context gathering module.

Example 8.2. Let us take the maps application as an example of the behavior of the context gathering module. To facilitate the way the context gathering module works, we will only take into account the CoPN composed of contexts PRIVATE, and POSITIONING and an exclusion dependency relation between them (Figure 6.3). Assume that initially the POSITIONING context is active, and suppose that an insecure connection is detected. This information triggers the activation of the PRIVATE context, sending an `@activate`(PRIVATE) message to the context manager module. Since the POSITIONING context is already active, the verification of the requested activation yields an inconsistent state and the deactivation is denied. In this case, a message is sent to the context gathering module stating that the POSITIONING context denies the activation of the PRIVATE context. This information is stored in the `errorState` variable of the PRIVATE context instantiated int he context gathering module. The usefulness of the `errorState` variable is to avoid constantly sending activation and deactivation messages to the context manager module. In this particular case, this means not sending the `@activate`(PRIVATE) message, even though the situation of the surrounding execution environment signals that the context should be activated.

Suppose now that the POSITIONING context is now deactivated (for example because the device looses connectivity). Since there are no context dependency relations constraining the deactivation of POSITIONING, the context is deactivated. As a result of this operation, the change of state of this context is broadcasted to all contexts instantiated in the context gathering module. In particular, the `errorState` variable of the PRIVATE context is modified with this information. Since the restriction impeding PRIVATE to be activated is removed, and the application is still in a situation where the private context should be activated, the `@activate`(PRIVATE) message is send to the context

manager, yielding the activation of this context.

Activation of contexts is interleaved with the base code of the system in most COP languages [103], harnessing the flexibility requirement of Dynamically Adaptive Software Systems (**D.3**). To avoid such problems, the context discovery component of the context-awareness architecture can be seen as an external component of the system, both physically and logically. We use the Bonjour communication protocol for all message exchanges between the context gathering and the context manager modules.

8.2 CoPN Tool Support for COP

Alongside the development of the CoPN programming model, we also focussed our development efforts in a simulation tool as a means to test the dynamics of context activations. There are three reasons motivating the development of this tool. First of all, we begin with the observation that Petri nets are dynamic models that provide a firsthand view about the state and possible actions of the system. As such, a (visual) representation could be useful to support the interaction requirement **M.1** for Dynamically Adaptive Software Systems. Secondly, a simulation tool that allows to activate and deactivate all contexts defined in the system could be used as a complement to the analysis of the system presented in Chapter 7. Finally, we note that testing Dynamically Adaptive Software Systems is a challenging task [159]. Up to this point, the only means to test context activations proposed in COP languages has been through hardcoding of context activations within the base logic of the system. This approach is neither maintainable nor scalable.

In this section we describe the main features of the current incarnation of our CoPN-IDE simulation tool[2] for the context-aware maps application motivated in Section 2.3.3. The CoPN-IDE is a multi-touch visualization tool for CoPNs. The tool eases the definition of contexts and context dependency relations at design time by allowing to directly manipulate the different components of the CoPN. For example, the addition of full contexts (i.e., instances of Figure 6.1), or individual transitions and arcs, and addition of tokens to particular places (context or temporary places). To ease the integration with existing Subjective-C applications, the tool allows the use of the DSL language described in Table 6.2 for the definition of contexts and context dependency relations.

Figure 8.7 shows the general view of the CoPN-IDE simulation tool. The top panel, labeled CoPN view, is the interactive multi-touch view of the complete system. The different elements of the CoPN (places and transitions) can be used to manually add tokens into places or fire the external transitions by long-pressing the corresponding element. In addition the two lower views of the

[2]The simulation tool is available for download at: `http://released.info.ucl.ac.be/Tools/Context-PetriNets`.

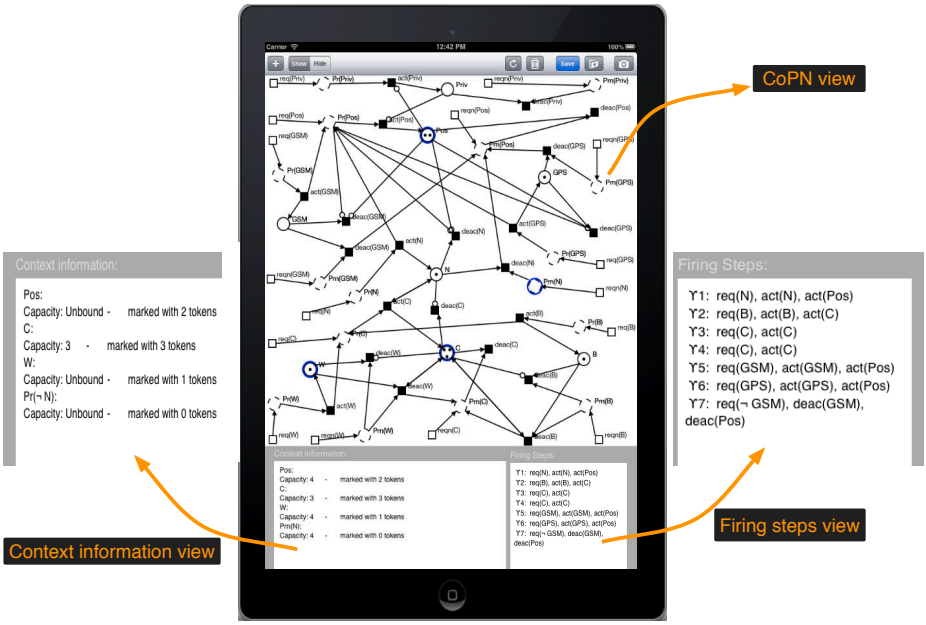


Figure 8.7: General view of the CoPN-IDE tool.

figure are used to display additional information about the CoPN places (left-hand side), and the firing steps of the different external transitions triggered by the user (right-hand side). The context information view provides an overview of contexts, their capacity and the tokens associated with it. The firing steps view provides a list of all sequences of fired transitions. This view only displays those firings of external transitions that lead to a consistent state of the system. We now explain additional functionality provided by the simulation tool, using the CoPN defined in Figure 7.2 as an example.

Figure 8.8 provides the visualization of the basic functionality of the interactive view: transition firing. External transitions are fired by long pressing them. In order to fire a transition, the color of the transition firing must be specified (i.e., the color of tokens), as shown in Figure 8.8. Up to this point we have only discussed a simplified version of CoPNs, in which it is only allowed to have tokens of one color. Section 9.2 shows the extension of the model to allow tokens of multiple colors in the system.

Firing each individual transition can be cumbersome and time consuming in order to reach a particular state in a large system. To further test interaction of the activation and deactivation of contexts we allow to manually define particular states of the system by manually adding tokes to specific places. Adding tokens to places can be used to further find incoherences of the CoPN by means of simulation, for example states in which other contexts are no longer reachable, or to test the behavior of

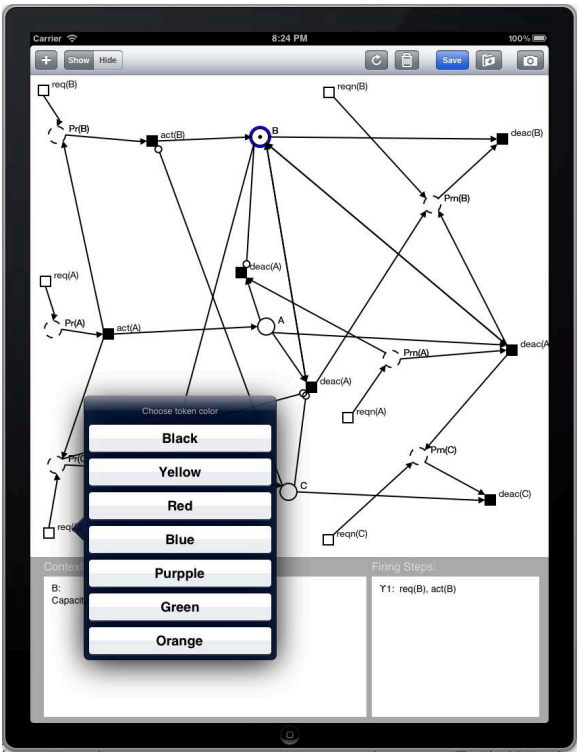


Figure 8.8: Manual firing of transitions with token colors.

inconsistent states. As in the case of transition firing, adding tokens to a place must specify the color of the token. Tokens are added by long pressing the place they are to be added to.

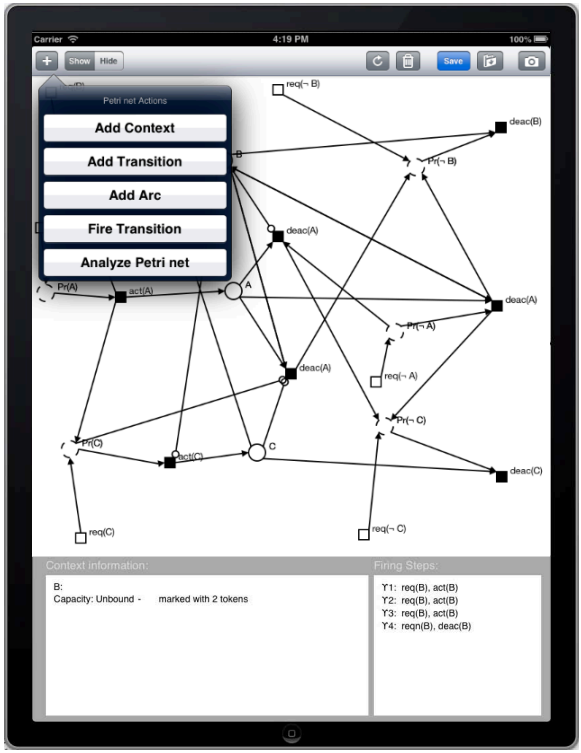


Figure 8.9: Edition and analysis of CoPNs menu.

Figure 8.9 shows the options to manipulate the structure of an existing CoPN. As mentioned before it is possible to add different components to the model, contexts, transitions, and arcs. In addition, by long pressing each of the components it is possible to modify their state, for example, changing the type of an arc from normal to inhibitor. Additionally, the fire transition option allows to fire external transitions without requiring the visualization of the CoPN.

The analysis option provided in the system's menu allows to analyze the currently displayed CoPN as described in Chapter 7. Selection of the analysis menu option automatically generates the Petri net with place capacities and without inhibitor arcs and all the test cases for the defined CoPN. The definition of the Petri net with place capacities and the analysis cases files are saved in the analysis file view of the simulation tool.

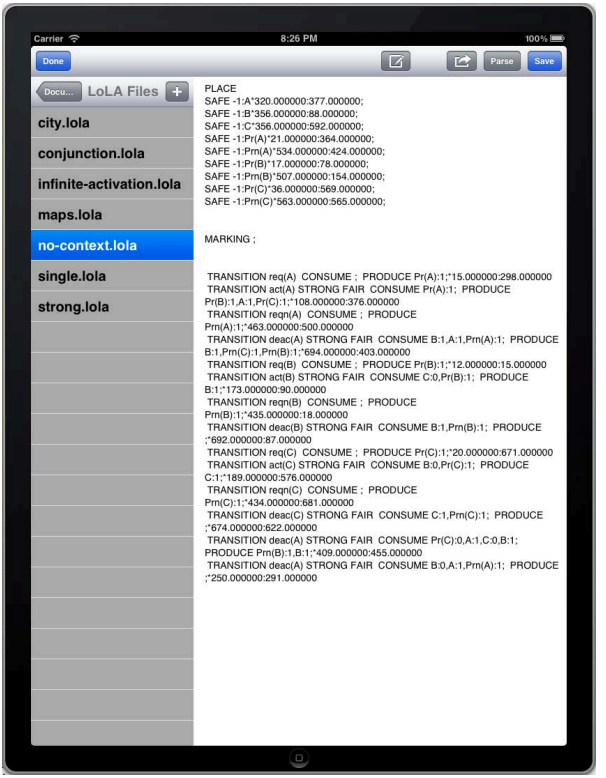


Figure 8.10: File view manager of the CoPN simulation tool.

Figure 8.10 shows the files manager view for the CoPN-IDE tool. This view is used to keep different definition of contexts and CoPNs. The CoPN-IDE currently supports two types of files, CoPN definition files (given by the DSL described in Table 6.2) and LoLA formatted files. CoPN definitions can be saved and loaded from these files.

As already mentioned, CoPNs can become cluttered in the visualization of systems with many contexts and context dependency relations between them. This problem becomes even bigger when the tool is used on small screen devices. For this reason, CoPN-IDE makes it possible to analyze or simulate context activations without relying on the visualization of the full CoPN.

8.3 Conclusion

This chapter describes the internals of CoPNs as a comprehensive programming model for Dynamically Adaptive Software Systems. In particular it provides the details of the way in which our model complies with the general architecture envisioned for such systems. The purpose of presenting CoPNs as a comprehensive programming model for Dynamically Adaptive Software Systems is to show the validity of CoPNs for such systems. Additionally, we want to ease the development of Dynamically Adaptive Software Systems, where the contexts and their different representations (e.g., context gathering module, context representation module) cooperate together under the same programming abstractions. A comprehensive programming model is also beneficial to ease the integration between adaptations and the base logic of the system.

With these objectives in mind, we revisited the architecture for context-awareness providing a finer-grained view of its different components. Figure 8.11 shows the different components of the context-awareness architecture overlaid over the programming model of CoPNs. Most of the components of the context-awareness architecture in the revisited model are comprised by the programming model of CoPNs. The work of this dissertation focuses on the management of inconsistencies. Although CoPN constitutes the first effort in offering a comprehensive programming model, we note that this is only a first iteration over the complete architecture of context-awareness and different modules must be extended, refined, or completed in order to truly have a comprehensive programming model for Dynamically Adaptive Software Systems.

The definition of the `ContextGathering` and `ContextAcquisition` modules introduced in CoPNs it is not longer necessary to know all context definitions and their interaction beforehand. The CoPN programming model allows us to introduce context definitions at run-time, and automatically integrate them with the run-time verification of consistency. This property complies with the independence requirement (**D.5**) of Dynamically Adaptive Software Systems.

During the definition of the CoPN programming model we identified a shortcoming regarding the testing of adaptations in existing COP languages, and Dynamically Adaptive Software Systems in general [159]. Testing of Dynam-

ically Adaptive Software Systems has already been proven challenging due to the multiplicity of possible adaptations of the system's behavior. COP is no exception to this, and only hardcoded techniques are used for the purpose of testing. In this chapter we also provide a first step towards the testing of Dynamically Adaptive Software Systems. Our approach consists of an interactive simulation tool that offers the possibility to define and modify COP systems. Additionally, the tool can be used to test the activation and deactivation of the contexts defined in the system, without requiring to tangle the context activations in the base logic of system. Additionally, the tool provides information about context's states and firing steps, as well as visualization of context activations. That information can be used to further identify if the context dependency relations defined between contexts behave as desired or not. The tool thus serves as complementary support for the identification of incoherent CoPN definitions.

To summarize, CoPNs successfully comply with the architecture proposed for Dynamically Adaptive Software Systems, providing explicit support for the different components of such architecture. In addition, the CoPN model offers a testing tool that can be used for the simulation of context activations. For these reasons we assert that CoPNs constitute a comprehensive programming model for Dynamically Adaptive Software Systems.

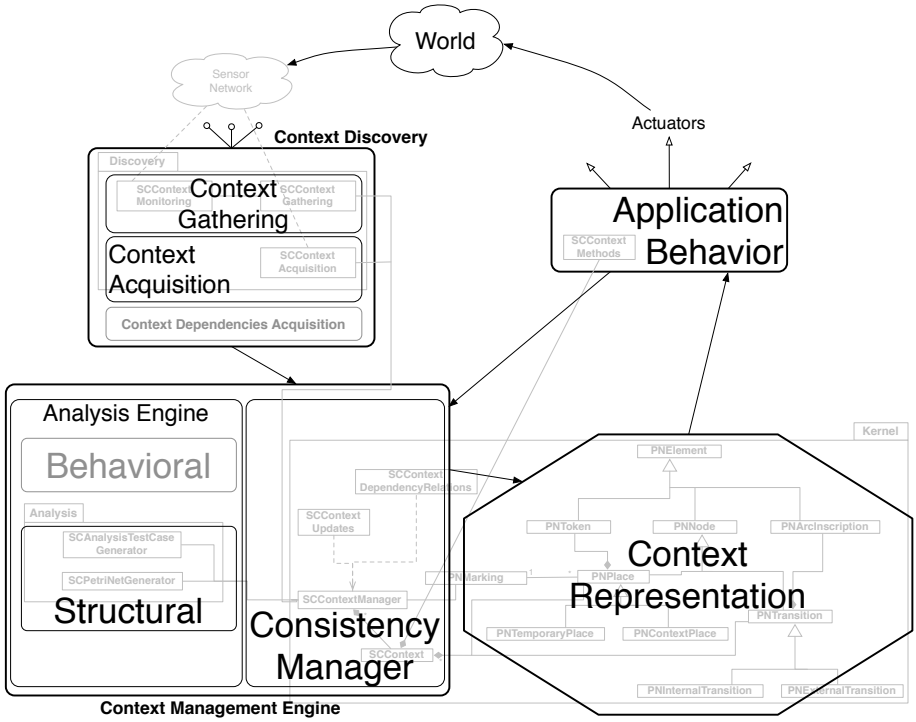


Figure 8.11: Refinement of the context-awareness architecture overlaid on the CoPN programming model.

Chapter 9

Context Petri Nets at Work

This chapter evaluates the programing model of CoPNs developed in this dissertation from three different viewpoints: the *usefulness* of the theory as a reasoning engine for Dynamically Adaptive Software Systems, the *appropriateness* of the theory to model Dynamically Adaptive Software Systems, and the *extensibility* of the theory to cover new aspects of such systems. In addition to these three viewpoints evaluating CoPNs as a formalism and as a programming model, we also provide a first assessment of the performance of managing the consistency of dynamic adaptations at run time.

The first two case studies presented in this chapter, respectively in Sections 9.1 and 9.2, reuse existing COP applications. In the first case study, the system is analyzed using CoPNs. The purpose of the analysis is to improve the application by removing all identified inconsistencies and incoherences. In the second case study, the system is used as a means to demonstrate the usefulness of CoPNs for the development of COP systems. The application was previously used to validate a new scoping mechanism of behavioral adaptations of COP. In the original implementation of the application the new scoping rules were realized using a programmatic approach. In Section 9.2 we obtained the same results by means of the formalism already defined for CoPNs, without requiring modification to the programming model or polluting the language syntax. The third case study, presented in Section 9.3, is an abstract case study used to evaluate how easy it is to extend the CoPNs formalism. The final section of this chapter evaluates the performance of CoPNs with respect to that of plain Subjective-C. Although we do not focus on performance in our work, we recognize that the run-time management of the system should not constrain its usability.

9.1 Analysis of Existing COP Systems

This section evaluates the usefulness of CoPNs as a reasoning engine for COP systems. To this end we use the Mobile City Guide, an application previously developed using Subjective-C [102]. We use CoPNs to reason about the configuration of contexts and the context dependency relations defined in the application. Based on the obtained results, we refactor the application to a configuration of contexts that does not present incoherences or inconsistencies.

9.1.1 Mobile City Guide

The Mobile City Guide is a mobile application that enables tourists visiting a city to navigate through the city's Points of Interest (POI). This case study is centered around the compositionality of CoPNs, and the analysis of the system's context definitions. The reason for using the Mobile City Guide to validate the analytic power of CoPN is twofold. On the one hand, being an existing application, the interactions defined between the contexts are unbiased towards the analysis of CoPNs. On the other hand, as shown in Figure 9.1, the Mobile City Guide contains examples of the four basic context dependency relations originally available in Subjective-C.

The Mobile City Guide application provides the possibility to create, customize and follow city tours based on a selection of POIs. The basic behavior of the application consists of three main features:

F.I. *Tour Creation & Selection:* This feature allows users to create and select city tours based on a list of available POIs. A tour can be followed in two modes: if the device has a GPSAntenna, there is a GuidedTour mode which guides users throughout the city according to a predefined route of POIs; in FreeWalk mode, users walk freely around the city and can view directions to the POIs they would like to visit.

F.II. *City Navigation:* In order to navigate through a city and its POIs, the application also provides the possibility of using a map, or compass navigation to navigate between POIs, which can for example be used in FreeWalk mode when no GPSAntenna is available.

F.III. *POI Display & Information:* POIs and their associated information can be displayed according to user-defined preferences, such as the preferred language of the user (UserLanguage), the age range of the visitor (TargetAudience), or the user's particular interests (UserInterests).

These basic features of the application can be further enhanced depending on the application's surrounding execution environment and user preferences. We now present the most important contexts that adapt the application's behavior.

1. A Language context allows to adapt the application's displayed information to a particular language, such as English or French, where two

language contexts cannot be active simultaneously, English□–□French. The language is either determined by the geographical location of the user, for example the UK or France, or it can be manually selected by the user in the application preferences (UserLanguage). We restrict the application to offer only one language at a time. These adaptations target the behavior of Feature **F.III**.

2. A Connectivity context allows us to fetch additional information about POIs, or to update current information whenever an internet connection is available. The availability of a particular connection protocol, Wifi enables the services associated with the internet connection, for example, Wifi–▶Connectivity. These adaptations target the behavior of Features **F.I** and **F.II**.

3. Images associated with POIs can by displayed according to the time of day at which the images were taken. Morning images are first shown before midday, while Night images are first shown at night. Clearly the two contexts cannot be active at the same time: Morning□–□Night. These contexts can also be used to modify the order in which POIs are displayed, for example, by ordering them according to their visiting hours. These adaptations target the behavior of Features **F.I** and **F.III**.

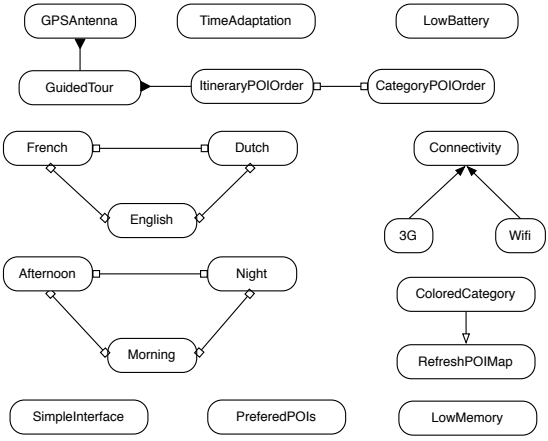


Figure 9.1: Context configuration diagram for the Mobile City Guide [102].

9.1.2 Analyzing the Mobile City Guide

We reason about the definition of the contexts and context dependency relations as originally defined for the Mobile City Guide to identify incoherences that might have been overlooked by the application designers. The reasoning process is supported by CoPNs and the LoLA. The original implementation of the

Mobile City Guide in Subjective-C consists of 20 contexts and 12 context dependency relations. As can be seem from Figure 9.1, the application consists of 10 isolated components, each of which is analyzed as an independent CoPN. Remember from Chapter 7 that properties to be verified about the system consist in the reachability and liveness of the CoPN. Since CoPNs consisting of a single context are never constrained for their activation and deactivation, we do not generate analysis test cases for such contexts (TIMEADAPTATION, LOWBATTERY, SIMPLEINTERFACE, PREFERREDPOIS, and LOWMEMORY in Figure 9.1) Furthermore, since Subjective-C does not allow developers to set a bound for contexts, all contexts are initially defined to be activated an undetermined number of times. Before diving into the analysis of the system using CoPNs, we first provide a bound for the contexts whenever this makes sense in the application domain.

Language adaptations are used in the Mobile City Guide to display information to users. However, we notice that it does not make sense to have multiple activations for a language; the application uses a language or it does not. Hence, the FRENCH, ENGLISH, and DUTCH contexts are each given a capacity of 1. Similarly, adaptations displaying images according to the time of day, only require a single activation for each context in the state space partition. In the Mobile City Guide, the AFTERNOON, MORNING, and NIGHT contexts should be activated at most once. To manage the interaction between these contexts, the Mobile City Guide manages their activation logic manually —that is, it is up to the programmer to ensure that the restrictions mentioned about these contexts are satisfied. Each of the pair-wise exclusion dependency relations defines the activation logic with respect to all the other connected contexts. The burden of managing the activation of these contexts gets reduced by giving a bound of 1 to each context in the CoPN.

The two CoPNs representing the pair-wise exclusion dependency relations between the language contexts and the time of day contexts are structurally identical, hence we treat them jointly in our discussion. For each of these two sets of exclusion dependency relations 13 test cases were generated by the CoPN model according to the specification given in Section 7.2.1: a general deadlock analysis, three reachability analyses, three state predicate analyses, and six liveness analyses. The analysis of each of these test cases using LoLA proved those CoPNs to be free of structural incoherences.

The GUIDEDTOUR and GPSANTENNA contexts are not bounded in the application domain, because these contexts can be activated from different situations in the surrounding execution environment. For example, activation may occur because of user preferences or application version (free vs. paying) in the case of the GUIDEDTOUR, or by means of using device's physical antenna or connecting to the antenna of external devices in the case of the GPSANTENNA. However, the order in which the POIs are displayed is unique. Hence, we give a capacity of 1 to the ITINERARYPOIORDER and CATEGORYPOIORDER. The generated capacity for the GPSANTENNA and GUIDEDTOUR contexts is 141.

For the CoPN composed of the contexts connected to the GUIDEDTOUR context 434 test cases are generated. 423 of these cases correspond to the live-

ness tests for each of the generated activation and deactivation transitions of contexts GuidedTour, GPSAntenna, and the activation transitions of context ItineraryPOIOrder.[1] The analysis of the state predicates defined for the requirement dependency relations, for example GuidedTour–◀GPSAntenna, detects that it is possible to reach a state m in which m(GuidedTour) ≥ 1 and m(GPSAntenna)=0. The output provided by LoLA gives a state and a path as follows:

```
STATE
  GPSAntenna0:1,
  GuidedTour1:1,
  CategoryPOIOrder0:1,
  ItineraryPOIOrder0:1
PATH
  req(GuidedTour)
  act(GuidedTour)0
```

As it was discussed in Section 7.2.1, this state cannot be reached. Remember from the definition of the test cases for a requirement dependency relation that the state predicate test case needs to be analyzed in conjunction with a liveness analysis of the transition deactivating the source context whenever the target is no longer active (deac(GuidedTour) in this example). The liveness analysis process produces as its output exactly the same state and path as the one produced by the state predicate analysis. Since the deac(GuidedTour) transition is an internal transition enabled for the illegal state, it must fire. As a consequence, the illegal state is never reached under CoPN semantics. Further analyzing of this information allows us to observed that it is not possible for the ItineraryPOIOrder to be activated unless the GuidedTour context is active. This means that it is not possible to order a set POIs whenever the user is in FreeWalk mode, which is not the expected behavior of the application. In the reminder of this chapter we refer to this situation (i.e., the behavior of the application is more restricted than what was expected by the user) as (INC.1). Although the CoPN is coherent, it presents an inconsistency in the observed behavior with respect to the predicted one.

For the CoPN composed of the contexts connected to the Connectivity context we note that contexts can be activated multiple times in the application domain. The generated capacity for the three contexts is 165. As a result the CoPN model generates 665 test cases, where 660 test cases correspond to the liveness analysis of the activation transitions of all three contexts in the CoPN and the deactivation transitions of the Connectivity context. All generated test cases for the implication dependency relation, described in Section 7.2.1, yield a satisfactory result. Hence, the CoPN associated with the Connectivity context and its related contexts is coherent.

Finally, we analyze the CoPN composed of the ColoredCategory and RefreshPOIMap contexts. In the application domain, the ColoredCategory context can only be activated once, while the RefreshPOIMap context could be activated multiple times. The generated capacity for the latter context is 249. For the analysis of the CoPN 754 test cases are generated, of which 749 correspond to the liveness test cases for the deactivation transitions of both contexts and the activation transitions of the RefreshPOIMap context. All generated test cases for the cause dependency relation,

[1] Currently the analysis of all test cases needs to be manually executed by programmers. However, must of these cases are equivalent to each other and thus would produce the same results. For example two activation transitions generated in the net unfolding.

described in Section 7.2.1, yield a satisfactory result. Hence, the CoPN composed of the ColoredCategory and RefreshPOIMap contexts is coherent.

In addition to the test cases generated by CoPNs, we produced three test cases to analyze the overall consistency of the system with respect to its predicted behavior. The test cases shown in the following all present an inconsistency with respect to the predicted behavior of the system.

(INC.2) Whenever GuidedTour and POIs information is displayed in a particular language, this language is either selected by the users as part of their preferences, or inferred by the system using the user's geographical position. The application is conceived such that the default behavior uses the selected language as the language to be displayed. However, if no language is defined in the users' preferences, the language is inferred from their position. We test the system to see if it is possible to yield a situation in which the GuidedTour context is active, but none of the language related contexts is. This state is expressed using the following state predicate:

```
FORMULA GuidedTour1 = 1 AND English0 = 1 AND Dutch0 = 1 AND French0 = 1
```

The state is reachable (taking into account both disconnected CoPNs), which should not be the case. The output generated by LoLA is as follows:

```
STATE
  Dutch0 : 1,
  French0 : 1,
  English0 : 1,
  GPSAntenna0 : 1,
  GuidedTour1 : 1,
  CategoryPOIOrder0 : 1,
  ItineraryPOIOrder0 : 1
PATH
  req(GuidedTour)
  act(GuidedTour)0
```

(INC.3) The purpose of the TimeAdaptation is to order and display the POIs according to the time of the day. However, this context does not have any connection with the Day, Afternoon, or Night contexts, which are in charge of representing the time of the day. Similarly to the previous case, we are interested in the reachability of a state where TimeAdaptation is active, and none of the other contexts is. The analysis provided by LoLA shows that indeed such a state is reachable. The output provided is similar to that of the previous case (INC.2).

(INC.4) The LowMemory context provides a behavioral adaptation that prohibits the application from downloading or updating information about the different tours or POIs when the device does not have sufficient storage space available. The behavioral adaptations associated with the Connectivity context allow the updating the information of tours and POIs in the system. These two contexts should not be active at the same time as their behavior is contradicting. We test if it is possible to reach a state in which they are both active at the same time using the state predicate:

```
FORMULA LowMemory > 0 AND Connectivity1 = 1
```

This illegal state is reachable. The output generated by LoLA is as follows:

```
STATE
  LowMemory : 1,
```

```
  Connectivity1 : 1,
  Pr(Connectivity)0 : 1
PATH
  req(LowMemory)
  act(LowMemory)
  req(Connectivity)0
  act(Connectivity)0
```

9.1.3 Revisiting the Mobile City Guide

The analysis of the ensemble of all independent CoPNs for each connected component of the Mobile City Guide application does not identify any incoherences as defined in Definition 7.2. However, it was possible to identify inconsistencies (INC.1) through (INC.4), which are due to accidental interactions between the defined contexts. In this section we revisit the Mobile City Guide application to refactor the interaction between its contexts in order to avoid inconsistencies currently existing in the system. After the modification, the application consists of 29 contexts and 31 context dependency relations. Figure 9.2 depicts the contexts and context dependency relations for the Mobile City Guide. We use the conjunction dependency relation ($\rightarrow$) introduced in Definition 6.17. Additionally, the figure also presents two new context dependency relations, a suggestion dependency relation ($\text{--}\triangleright$) and a disjunction dependency relation ($-\diamond$). The inner workings of these relations are introduced later in Section 9.3.

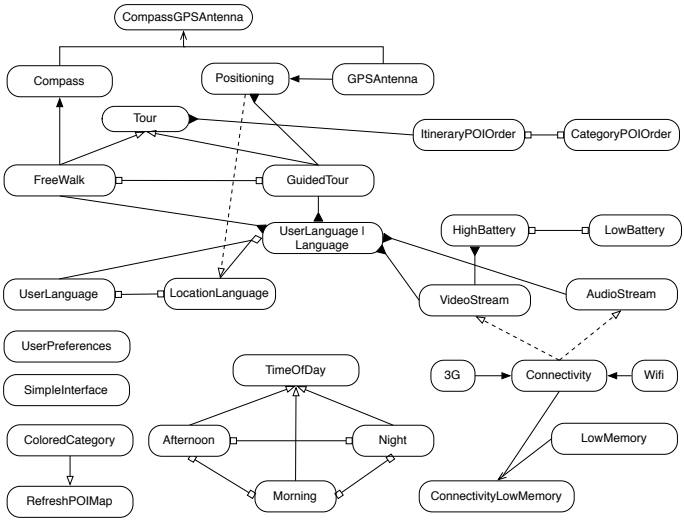


Figure 9.2: Refactored context configuration diagram for the Mobile City Guide.

Here we discuss the design choices used to solve the inconsistencies identified in the previous section.

(INC.1) The first inconsistency is addressed by applying three modifications to the context model. First of all, we introduce the FreeWalk context to manage

non-guided tours as part of the system's adaptations, rather than using it as the default behavior of the application. We turn the default behavior of the application to simply displaying the selected POIs in no particular order. Secondly, we abstracted the two existing options to follow a tour, GuidedTour and FreeWalk, by a new context Tour. Finally, the requirement dependency relation ItineraryPOIOrder–◀GuidedTour becomes ItineraryPOIOrder–◀Tour. These modifications of the model, allow the POIs to be ordered in both Tour modes. If there is no tour being followed, there is no need to order the POIs according to an itinerary. However, other orders, such as CategoryPOIOrder, may adapt the default behavior.

(INC.2) The second inconsistency is addressed by performing four modifications to the context model. First, we decouple the selected language by the user from the default behavior of the application, and add it as an independent context UserLanguage. Secondly, all other languages associated with a particular context, are abstracted as a Language context, and each particular language is treated as a parameter inside the Language context. Thirdly, an exclusion dependency relation (UserLanguage□–□Language) is defined between these two contexts. Fourth, the two language contexts are joined in a disjunction context UserLanguage||Language. This context is used as the entry point of all other contexts that require a language to communicate to the user. For example, VideoStream–◀UserLanguage|Language. Finally, we add a new Positioning context which is in charge of all services related to the user's geographical position. In particular it suggests the language to be used by the application, Positioning--▷Language. The inconsistency of the language behavior is avoided because there must be only one language active in the application. All functionality related to user interaction uses that language.

(INC.3) The third inconsistency is addressed by applying one modification to the context model. To solve this inconsistency we create a dependency relation between the different times of the day, Afternoon, Night, and Morning contexts, and the context abstracting all of them. We create causality dependency relations between each of these contexts and the TimeOfDay context, for example, Morning–▷TimeOfDay. The inconsistency is resolved now as the two contexts are activated simultaneously.

(INC.4) The fourth inconsistency is addressed by applying one modification to the context model. To avoid the contradicting behavior between the Connectivity and LowMemory contexts, we create a conjunction context composed of each of these contexts. The behavioral adaptations associated with the conjunction context ConnectivityLowMemory disambiguates any contradictory behavior that may exist whenever the two component contexts are active.

9.1.4 Evaluation

To evaluate the usefulness of CoPNs for the analysis of COP systems, we reused an existing application developed in Subjective-C and used CoPNs to test the definition of contexts and context dependency relations for incoherences and inconsistencies between the expected and observed behavior of the application.

In order to analyze the Mobile City Guide using CoPNs it was not required to manipulate the original definition of contexts and context dependency relations in

Subjective-C. Using the automatic generation of capacities and analysis test cases, CoPNs allow us to reason about COP system. From the analyses performed in Section 9.1.2, it was possible to observe that the generated analysis test cases provide a wide view on possible incoherent definitions of contexts. As a matter of fact, these test cases also aided in the identification of an inconsistency in application behavior. Nonetheless, we do note the usefulness of defining *additional* test cases based on the domain knowledge and the expected behavior of the system. Using additional test cases for the analysis of the application it was further possible to identify inconsistencies in its behavior. Section 11.4.2 discusses how additional cases could be automatically generated to analyze inconsistent application behavior according to the different behavioral adaptations defined in the system.

Usefulness of CoPNs is also observed from the definition of activation bounds for specific contexts. In Subjective-C, whenever the activation of a context requires additional management to take into account its activation count, the programmer is required to manage the activation of the context manually —that is, to explicitly express the allowed and disallowed cases for the activation of the context. Using CoPNs it is possible to assign bounds to contexts when they are being defined. The context activation automatically manages the allowed and disallowed activation conditions given by the bound, for example in the case of the adaptations related to the TimeOfDay in Figure 9.2. Hence, the use of place capacities avoids the burden of managing activation state of contexts.

The revisited Mobile City Guide application is used as an example showing how the results provided by the analysis of the system can be used. The application was stripped of all incoherences, inconsistencies, and unnecessary context management code. The modifications of the application mostly required the modification of the definition of the contexts and context dependency relations. Only localized modifications to the code were required, for example, to remove the code managing the context activation, such modification are not shown as it is not the objective of the case study.

9.2 Flexible Scoping of Adaptations Using CoPNs

This section evaluates CoPNs with respect to their appropriateness as a formalism for COP systems. In particular this section demonstrates how the formal basis of CoPNs could be used to extend the programming model of COP systems.

Remember from Section 4.1.3 that one of the descriptive characteristics of COP systems is the scoping of adaptations (property COP.8). COP languages provide scoping of adaptations to a local thread of execution, or global to all threads of the application. Using CoPNs, we can extend the regular scoping mechanism of COP languages to allow global and local activations of a context. A differentiation between context activations is beneficial for enabling behavioral adaptations for particular threads or object instances. In this dissertation we adhere to the delimitation specificity property (COP.8) in the scoping characteristics of COP languages. Local activations are understood to be context activations taking place in a particular thread, while global activations are understood to be context activations taking place for all running threads in the system. A local context activation implies that a context and its associated behavioral adaptations are only available in the threads which the context is activated for. All other threads in the system are oblivious to the behavioral adap-

tations introduced by the context. An experiment to support local and global scoping of adaptations was already proposed in Subjective-C [153]. In this section we present an alternative implementation for local scoping of adaptations which subsumes that previously provided one in Subjective-C. The most important difference between the original implementation and the one presented here, is that CoPNs allow and manage the interaction between global and local contexts [32].

9.2.1 Moving Triangles Application

Before diving into the details of how global and local scoped adaptations are reified by means of CoPNs, we provide an example application that illustrates the required support for global and local context activations [153]. This application consists of a canvas on a mobile device's screen, onto which a set of triangles can be drawn.

The application behavior is given by an independent thread per triangle drawn, that autonomously moves it forward on the canvas. The direction in which a single triangle moves can be modified to one of four directions (Up, Down, Left, or Right), each of them represented by a context and specializing an abstract Direction context. The direction in which a triangle moves dictates the color of the triangle. A Color context is used in combination with the Direction context to give triangles a specific color. The application also allows logging the movement of a triangle on the canvas by means of the Log context. The Direction, Color, and Log behavioral adaptations should be independent for each triangle. Additionally, the application presents two behavioral adaptations that all existing triangles in the application should exhibit. A Dashed context draws all triangles with a dashed line instead of a solid line. A Pause context pauses the movement of all triangles. An example of the application state is shown in Figure 9.3.

The moving triangles application already shows the convenience of providing global behavioral adaptations (e.g., Pause triangles' movement) and local behavioral adaptations (e.g., Log a triangle movement). If only one of the two scoping mechanisms is used, either only local activation or only global activations, the behavior provided by the other technique must be manually supported by programmers, which can render the development of the system more complicated and harder to maintain.

9.2.2 Supporting Global and Local Adaptation Scoping

Up to this point we have only discussed how COP systems are structured, and how CoPNs allow us to manage and reason about these systems. All context activations and deactivations discussed so far are *global* context activations, as this is the scoping mechanism natively supported by Subjective-C. To introduce *local* contexts (i.e., contexts for which the associated behavioral adaptations only affect the behavior of a single thread of execution) we take advantage of the thread model provided by Objective-C and Colored Petri nets [99].

Colored Petri nets extend the basic Petri net model by adding colors to tokens (a color can be seen as the type of the token). Transitions may be enabled for a particular color, thus transition firing is modified to take into account the color of input and output tokens.

Definition 9.1. *A CoPN with support for global and local context activations is a CoPN as defined in Definition 6.1, in which black tokens represent global activations*

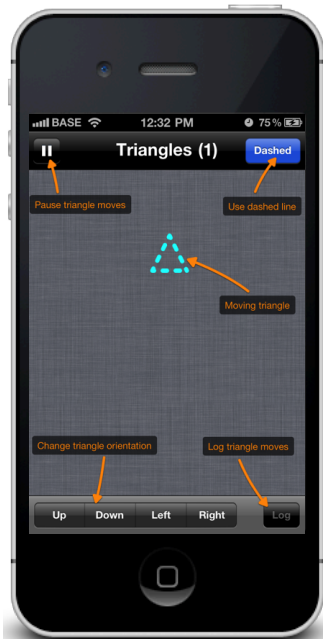


Figure 9.3: Screenshot of the moving triangles application [153].

and colored tokens represent local activations. In the following we refer to these CoPNs as ***scoped CoPNs****.*

The visual representation of a colored CoPN is illustrated in Figure 9.4 for the Log context of the moving triangles application.

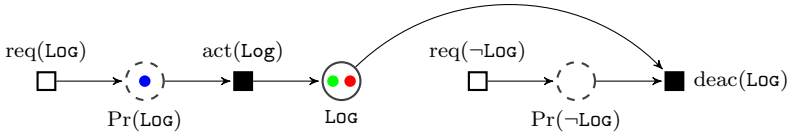


Figure 9.4: Three local activations for the Log context.

In scoped CoPNs global and local contexts can interact with each other. To account for such interaction, enabling and firing of transitions is modified as provided in the following definitions.

Definition 9.2. *In a scoped CoPN* $\mathcal{P} = \langle P_c, P_t, T_e, T_i, f, f_\circ, \rho, \mathcal{L}, m_0, \Sigma, \lambda \rangle$*, a transition* $t \in T_e \cup T_i$ *is enabled for a color* $l \in \mathcal{L}$ *at a reachable marking multiset* m *of* $\mathcal{P}$ *if and only if,* $\forall p \in {\bullet t},\ m(p,l) + m(p,black) > f(p,t,l)$ *and* $\forall p \in t^\circ,\ m(p,l) = 0 \wedge m(p,black) = 0$*.*

Definition 9.3. *In a scoped CoPN* $\mathcal{P} = \langle P_c, P_t, T_e, T_i, f, f_\circ, \rho, \mathcal{L}, m_0, \Sigma, \lambda \rangle$*, let* m *be a marking multiset, and* $t \in T_e \cup T_i$ *a transition such that* $m[t\rangle$*. A marking multiset*

m' is reachable from m by firing t, $m[t\rangle m'$, for a color $l \in \mathcal{L}$ if and only if $\forall\, p_{out} \in t\bullet$, $m'(p_{out}, l) = m(p_{out}, l) + f(t, p_{out}, l)$ and $\forall\, p_{in} \in \bullet t$

$$m'(p_{in}, l) = \begin{cases} m(p, l') - m(p_{in}, l) & \text{if } m(p_{in}, l) < f(p_{in}, t, l), \forall l' \in \mathcal{L} \\ m(p_{in}, l) - f(p_{in}, t, l') & \text{if } l = black,\ \forall\, l' \in \mathcal{L} \\ m(p_{in}, l) - f(p_{in}, t, l) & \text{otherwise} \end{cases}$$

Note that the conditions for enabling and firing transitions in scoped CoPNs take into account two sets of colors from the marking multiset. This is to enable interaction between global and local context activations. For every context marked with a black token, its associated behavioral adaptations are made available for *all* running threads in the applicaiton. Each thread in the application is identified by a unique token color. Contexts are marked with a colored token if they were activated for a given thread, in such a case behavior adaptations associated with the context are only available whenever the thread used for the activation is running.

If a context is activated globally, because the condition about the surrounding execution environment is reified for all threads, then in particular, said conditions are also valid for a specific thread. If the conditions reifying a context activation are no longer valid globally, then said conditions are not valid for any thread, and the behavioral adaptations associated with the context are removed from all threads. This interaction between global and local context activations can be better explained intuitively using the analogy of color subtractive composition theory.[2] In color theory, the black color can be seen as the subtractive combination of all colors —that is, it is the result of mixing all colors. In scoped CoPNs global context activations (black tokens) can be seen as activations in all threads of the system (combining all colors). Contexts activations for particular threads, can depend or be influenced by global context activations, since black is any other color in particular.

Whenever context activations depend on other contexts through context dependency relations, the interaction between the two contexts takes into account the thread in which each of them is activated. If the activation of a context depends on a second context, the latter context can be active both globally, or in the same thread the former context is to be activated in. If the deactivation of a context generates de deactivation of a second context and the former context is being deactivated globally, the latter context is also deactivated even if it is only active in one specific thread. The interaction between global and local contexts in the presence of context dependency relations is put into practice in the following example.

Example 9.1. As an example of the interaction between local and global context activations let us take a requirement dependency relation between two contexts `C` and `D`, as shown in Figure 9.5.

Take the initial marking of the scoped CoPN as $m_0(\mathtt{D}, black) = 1$. Suppose now that context `C` is to be activated in a specific thread $thread_{red}$. After the firing of the req(`C`) external transition with a *red colored token*, the marking of the CoPN is m_1, where $m_1(\mathtt{D}, black) = 1$ and $m_1(\mathtt{C}, red) = 1$. Note that the act(`C`) internal transition is enabled at marking m_1 because the input place `D` is marked with a black token, and by subtractive composition also red. Hence, the transition must fire and it yields the marking m_2 where $m_2(\mathtt{D}, black) = 1$ and $m_2(\mathtt{C}, red) = 1$, as shown in Figure 9.5.

Starting from the state shown in Figure 9.5, if context `D` is deactivated globally the firing of the req($\neg$ `D`) external transition yields a marking m_3, where $m_3(\mathtt{D}, black) = 1$,

[2] `http://www.worqx.com/color/color_systems.htm`

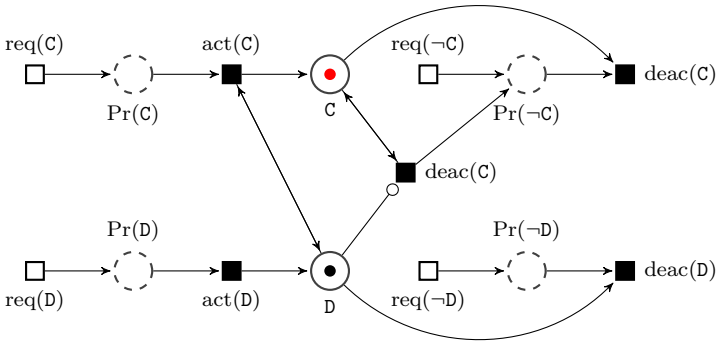


Figure 9.5: Global and local context activation interaction.

$m_3(Pr(\neg \mathtt{D}), black) = 1$, and $m_3(\mathtt{C}, red) = 1$. With this marking the deac(D) internal transition is enabled, and it fires. The firing of the transition leads to marking m_4, where $m_4(\mathtt{D}, l) = 0$, $\forall l \in \mathcal{L}$, and $m_4(\mathtt{C}, red) = 1$. Note that the bottommost internal transition labeled deac(C) becomes enabled because the deactivation is being performed globally, so, in particular, the *red* thread is also being deactivated. The firing of this transition leads to an empty marking. If context D is deactivated locally for the red thread, the process is equivalent.

In order to extend the scoping mechanism of COP systems using our formalism of scoped CoPN, as explained above, the selection mechanism of behavioral adaptations must be modified. At the language level, the context activation constructs are modified to take into account the specific threads for which a context is to be applicable. The constructs `@activate`(*context* `in` *thread*) and `@deactivate`(*context* `in` *thread*) are introduced with this purpose, where *thread* is a running thread defined in the application. For example, in Figure 9.5 the `Log` context has been activated for two particular threads. One thread tG associated to color green, and one thread tR associated to color red. Respectively, these activations are `@activate`(`Log in` tG) and `@activate`(`Log in` tR). Additionally, the selection mechanism of behavioral adaptations is modified to take into account the activation thread of each context. Scoped CoPNs reuse the method dispatching mechanism introduced in Subjective-C to deal with local and global context activations and method calls [153].

Snippet 9.1 shows a code extract of the moving triangles application that illustrates how global and local contexts are used. In the snippet we assume to have two triangles: triangle `tG` running on thread `tTG`, and triangle `tB` running on the thread `tTB`.

Triangles start to move independently based on their direction. If we want to log the movements for triangle `tG`, it is necessary to activate the `Log` context in the thread in which the triangle is running (Line 9 in Snippet 9.1). After the context has been activated (supposing the activation does not yields an inconsistency) all movements of triangle tG are logged. However, the movements of triangle tB remain unlogged. To modify the behavior of all moving triangles, we can use a global adaptation as the one shown in Line 11, which draws all triangles with a dashed line, since the context activation took place globally.

```
1  //Definition of the two triangles
2  Triangle *tG = [[Triangle alloc] initWithCenterPoint:aPoint movingBounds↩
       :[trianglesView bounds]];
3  NSThread *tTG = [[NSThread alloc ]initWithTarget:tG selector:@selector(↩
       startMoving:) object:nil];
4  tG.movingThread = tTG;
5  Triangle *tB = [[Triangle alloc] initWithCenterPoint:aPoint movingBounds↩
       :[trianglesView bounds]];
6  NSThread *tTB = [[NSThread alloc ]initWithTarget:tB selector:@selector(↩
       startMoving:) object:nil];
7  tB.movingThread = tTB;
8  //Movement of first triangle is logged
9  @activate(Log in tTG);
10 //Both triangles drawn with dashed lines
11 @activate(Dashed);
```

Snippet 9.1: Effect of global and local context activations from the language perspective.

9.2.3 Evaluation

To evaluate the appropriateness of CoPNs for the extension of COP systems, we used colored CoPNs, a mechanism to support local and global scoping of context activations. Moreover, we validated the approach by means of an application initially developed in Subjective-C and rebuilt it using the CoPN implementation of Subjective-C.

In order to rebuild the Subjective-C application with CoPNs, it was sufficient to modify the definitions of the contexts and context dependency relations to use the DSL language of Table 6.1. To introduce local and global context activations, Subjective-C explicitly differentiates between local and global contexts —that is, a contexts that can be activated either locally or globally, but not with both scoping mechanisms. This implies that in Subjective-C's DSL it must be specified which contexts are local and which contexts are global. A consequence of this design choice, is that global and local contexts do not interact with each other.

In CoPNs, we opted for allowing interaction between local and global contexts activations for two reasons. First of all, interaction between local and global contexts renders systems more flexible and dynamic. For example, the DASHED context of the moving triangles application can be activated for a single triangle. If context activations are kept independent, it is necessary to generate more event messages for context activation, which may not always make sense. For example, if a context is made inactive globally, why should events be generated to make it inactive locally. Secondly, differentiating between local and global contexts pollutes the definition and reasoning about contexts. Using CoPNs, it is possible to provide local and global context activations simply by giving a semantics to the marking multiset by means of colored tokens. The run-time management of the system required minimal modification to adhere to the semantics of the interaction between global and local context activations.

The local or global activation of contexts in Subjective-C is tightly coupled with the notion of threads. However, local or global context activation can take place with respect to objects and classes, or local and remote services. To adapt the concepts of local and global activations to any of these ideas, the complete Subjective-C scoping and selection mechanisms have to be revisited. Using scoped CoPNs, on the contrary, only the conditions in which each local or global activation occurs need to be changed. The logic of the activation of contexts and their interactions remains

valid. Section 11.4.4 expands the discussion about the selection mechanism of CoPNs.

9.3 Extending Context Dependency Relations

In this section we evaluate the extensibility of CoPNs with respect to their modeling power and ease to define and express different interactions between contexts (i.e., context dependency relations). To illustrate this extensibility we provide in the following the definition of two new context dependency relations using CoPNs, namely disjunction and suggestion, respectively represented by the symbol types $\vee$ and S.

9.3.1 Disjunction

We first introduce the **disjunction** context dependency relation. Similarly to the conjunction dependency relation introduced in Definition 6.17, the disjunction dependency is not manipulated directly, but its activation and deactivation depend on the activation state of its related contexts.

A disjunction dependency relation can be defined between any number of contexts, used to represent situations in which the same behavioral adaptation may be appropriate to different contexts, and hence, is made available when any of such contexts becomes active. The disjunction dependency relation is implemented in the EventCJ COP language as part of the composite layers abstraction [104, 105], and it follows the semantics of the logical *or* ($\vee$) operator.

The disjunction dependency relation gathers the interaction between a set of contexts. To mange such interaction we introduce a new context place, p' labeled $\mathtt{A}_1|\cdots|\mathtt{A}_n$, representing the contexts involved in the dependency relation as a unit. The p' context place cannot be directly manipulated —that is, it cannot be activated by firing external transitions. Instead the context place is automatically activated whenever one of the source contexts involved in the dependency relation $\mathtt{A}_j$ become active. As a consequence, if all the contexts $\mathtt{A}_j$ are inactive, it must be ensured that the new context place p' is not marked.

Definition 9.4 (Disjunction). *The disjunction dependency relation for a set of contexts ($\multimap(\mathtt{A}_1,\ldots,\mathtt{A}_n)$) is defined as a tuple $\langle\vee,\mathcal{C}_{A_1},\ldots,\mathcal{C}_{A_n}\rangle$, such that for $1 \leq j \leq n$, the singleton CoPNs $\mathcal{C}_{A_j}$ are pairwise different. The composed CoPN defining the disjunction dependency relation, $\mathcal{P} = \langle P_c, P_t, T_e, T_i, f, f_\circ, \rho, \mathcal{L}, m_0, \Sigma, \lambda\rangle$, is obtained by the union of all singleton CoPNs, $\mathcal{P} = \mathtt{union}(\{\mathcal{C}_{A_1},\ldots,\mathcal{C}_{A_n}\})$, and the application of the $\mathtt{ext}_\vee$ and $\mathtt{cons}_\vee$ functions to $\mathcal{P}$.*

The $\mathtt{ext}_\vee$ function is defined as $\mathtt{ext}_\vee(\mathcal{P},\langle\vee,\mathcal{C}_{A_i},\ldots,\mathcal{C}_{A_n}\rangle) = \mathcal{P}'$, between the singleton CoPNs $\mathcal{C}_{A_j} = \langle P_{c_{A_j}}, P_{t_{A_j}}, T_{e_{A_j}}, T_{i_{A_j}}, f_{A_j}, f_{\circ_{A_j}}, \rho_{A_j}, \mathcal{L}_{A_j}, m_{0_{A_j}}, \Sigma_{A_j}, \lambda_{A_j}\rangle$, such that for $1 \leq j \leq n$, $\mathcal{C}_{A_j} \subset \mathcal{P} = \langle P_c, P_t, T_e, T_i, f, f_\circ, \rho, \mathcal{L}, m_0, \Sigma, \lambda\rangle$ and $\mathcal{P}' = \langle P'_c, P'_t, T_e, T'_i, f', f_\circ, \rho', \mathcal{L}, m_0, \Sigma', \lambda'\rangle$, where:

$$\Sigma' = \Sigma \cup \{\mathtt{A}_1|\cdots|\mathtt{A}_n, Pr(\mathtt{A}_1|\cdots|\mathtt{A}_n), Pr(\neg\mathtt{A}_1|\cdots|\mathtt{A}_n), act(\mathtt{A}_1|\cdots|\mathtt{A}_n), deac(\mathtt{A}_1|\cdots|\mathtt{A}_n)\}$$

$$P'_c = P_c \cup \{p'\}$$
$$P'_t = P_t \cup \{p'', p'''\}$$
$$T'_i = T_i \cup \{t', t''\}$$

$$\lambda'(e) = \begin{cases} \lambda(e) & \text{if } e \in P_c \cup P_t \cup T_e \cup T_i \\ \mathtt{A}_1|\cdots|\mathtt{A}_n & \text{if } e = p' \\ Pr(\mathtt{A}_1|\cdots|\mathtt{A}_n) & \text{if } e = p'' \\ Pr(\neg\mathtt{A}_1|\cdots|\mathtt{A}_n) & \text{if } e = p''' \\ act(\mathtt{A}_1|\cdots|\mathtt{A}_n) & \text{if } e = t' \\ deac(\mathtt{A}_1|\cdots|\mathtt{A}_n) & \text{if } e = t'' \end{cases}$$

$$\rho'(t) = \begin{cases} \rho(t) & \text{if } t \ in T_e \cup T_i \\ 2 & \text{if } t = t' \vee t = t'' \end{cases}$$

$$f'(p,t,l) = \begin{cases} f(p,t,l) & \text{if } p \in P_c \cup P_t \wedge t \in T_e \cup T_i \wedge l \in \mathcal{L} \\ 1 & \text{if } p = p'' \wedge t = t' \wedge l \in \mathcal{L} \\ 1 & \text{if } p = p''' \wedge t = t'' \wedge l \in \mathcal{L} \\ 1 & \text{if } p = p' \wedge t = t'' \wedge l \in \mathcal{L} \\ 0 & \text{otherwise} \end{cases}$$

$$f'(t,p,l) = \begin{cases} f(t,p,l) & \text{if } p \in P_c \cup P_t \wedge t \in T_e \cup T_i, \wedge l \in \mathcal{L} \\ 1 & \text{if } t = t' \wedge p = p' \wedge l \in \mathcal{L} \\ 0 & \text{otherwise} \end{cases}$$

$$f'_\circ(p,t) = \begin{cases} f_\circ(p,t) & \text{if } p \in P_c \cup P_t \wedge t \in T_e \cup T_i \\ 0 & \text{otherwise} \end{cases}$$

The $\mathtt{ext}_\vee$ *function introduces a new context place to represent the activation state of any of the component contexts —that is, the context place is marked whenever one of the context involved in the disjunction dependency relation is active. The introduced internal transitions and temporary places are used to manage the activation and deactivation of the new context place.*

The $\mathtt{cons}_\vee$ *function is defined as* $\mathtt{cons}_\vee(\mathcal{P}, \langle \vee, \mathcal{C}_{A_1}, \ldots, \mathcal{C}_{A_n} \rangle) = \mathcal{P}'$*, for the singleton CoPNs* $\mathcal{C}_{A_j} = \langle P_{c_{A_j}}, P_{t_{A_j}}, T_{e_{A_j}}, T_{i_{A_j}}, f_{A_j}, f_{\circ_{A_j}}, \rho_{A_j}, \mathcal{L}_{A_j}, m0_{A_j}, \Sigma_{A_j}, \lambda_{A_j} \rangle$*, such that for* $1 \leq j \leq n$*,* $\mathcal{C}_{A_j} \subset \mathcal{P}$ *and* $\mathcal{P}' = \langle P_c, P_t, T_e, T_i, f', f_\circ, \rho, \mathcal{L}, m_0, \Sigma, \lambda \rangle$*, where* $\forall\, p \in P_c \cup P_t$*,* $\forall\, t \in T_e \cup T_i$ *and* $\forall\, l \in \mathcal{L}$*,*

$$f'(p,t,l) = f(p,t,l)$$

$$f'(t,p,l) = \begin{cases} 1 & \text{if } \mathtt{A}_j \in t\bullet \wedge \mathtt{A}_j \notin \bullet t \text{ for } 1 \leq j \leq n \\ & \wedge\, p = p'' \wedge l \in \mathcal{L} \quad (9.1) \\ 1 & \text{if } \mathtt{A}_j \in t\bullet \wedge \mathtt{A}_j \notin \bullet t \text{ for } 1 \leq j \leq n \\ & \wedge\, p = p''' \wedge l \in \mathcal{L} \quad (9.2) \\ f(t,p,l) & \text{otherwise} \quad (9.3) \end{cases}$$

The $\mathtt{cons}_\vee$ *function adds arcs from the activation and deactivation transitions of each of the contexts composing the disjunction dependency relation. Every time one of such contexts is activated, the disjunction context place introduced in the* $\mathtt{ext}_\vee$ *function is requested for activation. Similarly, every time one of the component contexts is deactivated, the disjunction context place introduced by the* $\mathtt{ext}_\vee$ *function is requested for deactivation. As a consequence, the disjunction context place* always *has as many tokens as there are tokens in all of the component contexts of the dependency relation.*

Figure 9.6 illustrates the CoPN representing the disjunction dependency relation $\langle \vee, \mathcal{C}_{\mathtt{U}}, \mathcal{C}_{\mathtt{L}} \rangle$ between the contexts UNFOCUSED (U) and LOWBATTERY (L) introduced in Example 9.2.

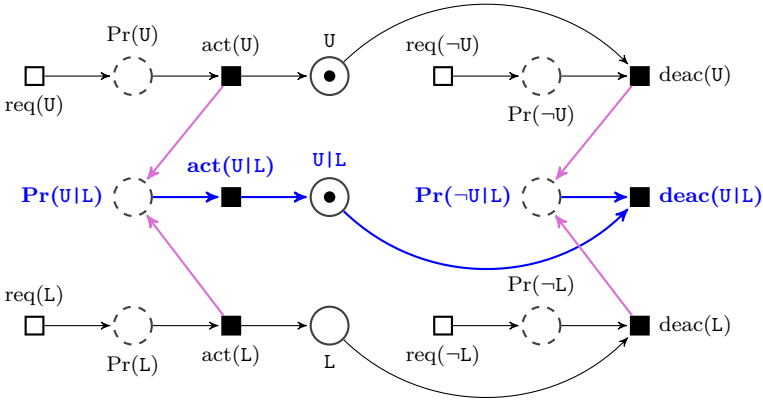


Figure 9.6: Disjunction dependency relation ($-\diamond$ (U L)).

Example 9.2. We describe the behavior of the disjunction dependency relation in CoPNs by means of an example first motivated by Kamina et al. [104]. As an example we use a multi-tabbed message board system. Each tab in the system displays the record of the messages posted on the message board. The message board should be updated whenever a message is posted. With the multi-tab view, only the current tab in focus is updated frequently, unfocused tabs are updated infrequently. Additionally, in case the device accessing the message board is running low on battery, all of the tabs are set to update infrequently in order to save battery. In this case an alert message is displayed to the user.

For a particular tab, the adaptation managing the infrequent update behavior can be modeled using a disjunction dependency relation. The components of the disjunction are the conditions in which updates are supposed to occur infrequently, namely UNFOCUSED (U) and LOWBATTERY (L). The disjunction dependency relation, $\langle \vee, \mathcal{C}_{\mathtt{U}}, \mathcal{C}_{\mathtt{L}} \rangle$, between the two contexts is shown in Figure 9.6.

For example if the tab t goes out of focus, $req(\mathtt{U})$ is triggered for that tab.[3] Since the internal transition $act(\mathtt{U})$ becomes enabled, it fires adding a token to context U and forwarding the activation to $Pr(\mathtt{U|L})$. In this case $act(\mathtt{U|L})$ becomes enabled, and fires adding a token to U|L (the state shown in Figure 9.6). If the tab is focused again, the $req(\neg\mathtt{U})$ transition is fired. As $deac(\mathtt{U})$ fires, U becomes inactive, enabling $deac(\mathtt{U|L})$ (because L is not active). When such transition fires, none of the contexts in the CoPN are marked.

[3] EventCJ provides support for global and local context activations. This example uses that scoping mechanism for the activation state of each tab. However, the locality of activations is not important for the purpose of our example.

9.3.2 Suggestion

The second context dependency relation introduced is **suggestion**, an extension of the semantics of the causality dependency relation, presented in Definition 6.14. A suggestion dependency relation (A--▷B) between two contexts A and B is used to model the situation in which the activation of the source context A requests the activation of the target context B. The relation is a suggestion in the sense that A merely suggests the activation of B but if this context cannot be activated, A is still activated. Additionally, in case that context B is active, the deactivation of A requests the deactivation of B. The suggestion dependency relation was first proposed in the PhenomenalGem COP language [148].

The suggestion dependency relation cannot be represented in the CoPN model as is. This is due to the conditional constraint over the activation of the target context. Intuitively, the activation of the source context forwards the activation request to the target context. If the target context cannot be activated, for example, because of another context dependency relation, the temporary place representing the preparation to activate the target context is marked, yielding an inconsistent state. According to the activation semantics described in Section 6.3.1, the activation is rolled back, and none of the contexts are activated.

To circumvent this problem we take advantage of the priority function ρ for the prioritization of transition firings in the definition of CoPNs. We need a way to ensure that the temporary place $Pr()$ for the target context is emptied, hence, avoiding the inconsistent state. For this purpose, we define a "new" type of internal transition which we call **clearing transitions**. Clearing transitions have the same semantics as internal transitions —that is, they fire as soon as they become enabled. For purposes of clarity, we represent clearing transitions as the set T_c. Nonetheless, the reader must remember that clearing transitions *are* internal transitions $T_c \subset T_i$. However, we will give clearing transitions a firing priority lower than that of other internal transitions. Using the priority function, clearing transitions are characterized by their priority function as $T_c = \{t \in T_i \mid \rho(t) = 1\}$.

Notation 9.1.

- *Clearing transitions are labeled as cl(·)*

Note that introducing clearing transitions does not change the activation semantics of CoPN. In particular, Reduction rules 6.20 through 6.23 decide which transition to fire based on the priority of enabled transitions. That is, from the enabled transitions in the priority set Σ, those transitions with a highest priority are always fired before those with lower priority, ensuring that all internal transitions fire before any of the clearing transitions.

The suggestion dependency relation between two contexts A and B, is such that for every activation of the source context A, the activation of the target B is requested. However, if B cannot be activated, the activation of A still succeeds. The deactivation of A automatically triggers the deactivation of B, only if the later was previously active. If context B is inactive, context A is deactivated without requesting the deactivation of B. B can be activated and deactivated independently from A.

Definition 9.5 (**Suggestion**). *The suggestion dependency relation between two contexts (A--▷B) is defined as the tuple $\langle S, \mathcal{C}_A, \mathcal{C}_B \rangle$, where $\mathcal{C}_A$ and $\mathcal{C}_B$ are two different singleton CoPNs. The composed CoPN defining a suggestion dependency relation,*

$\mathcal{P} = \langle P_c, P_t, T_e, T_i, f, f_\circ, \rho, \mathcal{L}, m_0, \Sigma, \lambda\rangle$ *is obtained by the union of the singleton* CoPNs, $\mathcal{P} = \mathbf{union}(\{\mathcal{C}_A, \mathcal{C}_B\})$, *and the application of the* $\mathbf{ext}_S$ *and* $\mathbf{cons}_S$ *functions to* $\mathcal{P}$.

The $\mathbf{ext}_S$ *function, similar to the* $\mathbf{ext}_C$, *is defined as* $\mathbf{ext}_S(\mathcal{P}, \langle S, \mathcal{C}_A, \mathcal{C}_B\rangle) = \mathcal{P}'$, *between two singleton* CoPNs $\mathcal{C}_A = \langle P_{c_A}, P_{t_A}, T_{e_A}, T_{i_A}, f_A, f_{\circ_A}, \rho_A, \mathcal{L}_A, m_{0_A}, \Sigma_A, \lambda_A\rangle$ *and* $\mathcal{C}_B = \langle P_{c_B}, P_{t_B}, T_{e_B}, T_{i_B}, f_B, f_{\circ_B}, \rho_B, \mathcal{L}_B, m_{0_B}, \Sigma_B, \lambda_B\rangle$, *where* $\mathcal{C}_A, \mathcal{C}_B \subset \mathcal{P} = \langle P_c, P_t, T_e, T_i, f, f_\circ, \rho, \mathcal{L}, m_0, \Sigma, \lambda\rangle$ *and* $\mathcal{P}' = \langle P_c, P_t, T_e, T_i', f', f_\circ', \rho', \mathcal{L}, m_0, \Sigma', \lambda'\rangle$, *such that:*

$$\Sigma' = \Sigma \cup \{cl(Pr(\mathtt{B}))\}$$

$$T_i' = T_i \cup \{t', t''\}$$

$$\lambda'(e) = \begin{cases} \lambda(e) & \text{if } e \in P_c \cup P_t \cup T_e \cup T_i \\ deac(\mathtt{A}) & \text{if } e = t' \\ cl(Pr(\mathtt{B})) & \text{if } e = t'' \end{cases}$$

$$\rho'(t) = \begin{cases} \rho(t) & \text{if } t \in T_c \cup T_i \\ 2 & \text{if } t = t' \\ 1 & \text{if } t = t'' \end{cases}$$

$$f'(t,p,l) = \begin{cases} f(t,p,l) & \text{if } t \in T_e \cup T_i \ \wedge\ p \in P_c \cup P_t \ \wedge\ l \in \mathcal{L} \\ 0 & \text{otherwise} \end{cases}$$

$$f'(p,t,l) = \begin{cases} f(p,t,l) & \text{if } p \in P_c \cup P_t \ \wedge\ t \in T_e \cup T_i \ \wedge\ l \in \mathcal{L} \\ 1 & \text{if } \lambda(p) = \mathtt{A} \ \wedge\ t = t' \ \wedge\ l \in \mathcal{L} \\ 1 & \text{if } \lambda(p) = Pr(\neg\mathtt{A}) \ \wedge\ t = t' \ \wedge\ l \in \mathcal{L} \\ 1 & \text{if } \lambda(p) = Pr(\mathtt{B}) \ \wedge\ t = t'' \ \wedge\ l \in \mathcal{L} \\ 0 & \text{otherwise} \end{cases}$$

$$f_\circ'(p,t) = \begin{cases} f_\circ(p,t) & \text{if } p \in P_c \cup P_t \ \wedge\ t \in T_e \cup T_i \\ 1 & \text{if } \lambda(p) = \mathtt{B} \wedge t = t' \\ 0 & \text{otherwise} \end{cases}$$

The newly introduced by the $\mathbf{ext}_S$ *function,* t' *labeled* $deac(\mathtt{A})$, *manages the interaction for the deactivation of the source context whenever the target context is inactive. This case can arise through the deactivation of the target context, since it is independent of the source context. Transition* t'', *labeled* $cl(Pr(\mathtt{A}))$, *is introduced to remove a request to activate* B *that cannot be processed. This transition is the last internal transition to fire because it has the lowest priority. The use of this* clearing *transition is to allow the activation of the source context* A, *even if the target context* B *cannot be activated.*

The $\mathbf{cons}_S$ *function, similar to* $\mathbf{cons}_C$, *is defined as* $\mathbf{cons}_S(\mathcal{P}, \langle C, \mathcal{C}_A, \mathcal{C}_B\rangle) = \mathcal{P}'$, *between two singleton* CoPNs $\mathcal{C}_A = \langle P_{c_A}, P_{t_A}, T_{e_A}, T_{i_A}, f_A, f_{\circ_A}, \rho_A, \mathcal{L}_A, m_{0_A}, \Sigma_A, \lambda_A\rangle$ *and* $\mathcal{C}_B = \langle P_{c_B}, P_{t_B}, T_{e_B}, T_{i_B}, f_B, f_{\circ_B}, \rho_B, \mathcal{L}_B, m_{0_B}, \Sigma_B, \lambda_B\rangle$, *where* $\mathcal{C}_A, \mathcal{C}_B \subset \mathcal{P} = \langle P_c, P_t, T_e, T_i, f, f_\circ, \rho, \mathcal{L}, m_0, \Sigma, \lambda\rangle$ *and* $\mathcal{P}' = \langle P_c, P_t, T_e, T_i, f', f_\circ, \rho, \mathcal{L}, m_0, \Sigma, \lambda\rangle$, *such that*

$$f'(p,t,l) = \begin{cases} 1 & \text{if } \lambda(p) = \mathtt{B} \ \wedge\ \mathtt{A} \in {}^\bullet t \ \wedge\ \mathtt{A} \notin {}^\bullet t \ \wedge \\ & \quad \mathtt{B} \notin t^\circ \ \wedge\ l \in \mathcal{L} \qquad (9.4) \\ f(p,t,l) & \text{otherwise} \qquad (9.5) \end{cases}$$

$$f'(t,p,l) = \begin{cases} 1 & \textit{if } \lambda(p) = \mathtt{B} \wedge \mathtt{A} \in {\bullet t} \wedge \mathtt{A} \notin {\bullet t} \wedge \\ & \mathtt{B} \notin {\circ t} \wedge l \in \mathcal{L} \quad (9.6) \\ 1 & \textit{if } \lambda(p) = Pr(\neg \mathtt{B}) \wedge \mathtt{A} \in {\bullet t} \wedge \mathtt{A} \notin t\bullet \wedge \\ & \mathtt{B} \notin {\circ t} \wedge l \in \mathcal{L} \quad (9.7) \\ 1 & \textit{if } \lambda(p) = Pr(\mathtt{B}) \wedge \mathtt{A} \in t\bullet \wedge \mathtt{A} \notin {\bullet t} \wedge l \in \mathcal{L} \quad (9.8) \\ f(p,t,l) & \textit{otherwise} \quad (9.9) \end{cases}$$

The arcs introduced by the $\mathtt{cons}_S$ *are used to forward the deactivation of the source context to the target context, whenever the source context is deactivated and the target context is active, Equations (9.4), (9.6), and (9.7). The behavior provided by these constraints complements the deactivation rule for the source context introduced by the* $\mathtt{ext}_S$ *function. Additionally, every activation of the source context requests the activation of the target context, given that the source context is not an input place of such transition, Equation* (9.8).

Figure 9.7 illustrates the CoPN representing the suggestion dependency relation $\langle S, \mathcal{C}_{\mathtt{M}}, \mathcal{C}_{\mathtt{Q}} \rangle$ between the MEETING (M) and QUIET (Q) contexts introduced in Example 9.3.

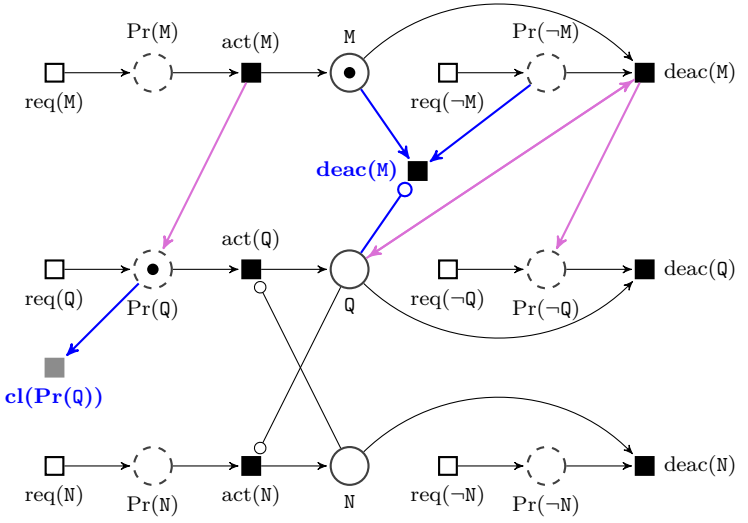


Figure 9.7: Suggestion dependency relation (M--▷Q) and exclusion dependency relation (Q□–□N).

Example 9.3. We describe the behavior of the suggestion dependency in CoPNs by means of the example motivated in PhenomenalGem. As an example motivating the need for the suggestion dependency relation, let us take the prototypical example of the context-aware phone forwarding system. In this scenario the user of a mobile device wishes to forward calls received during a MEETING (M) to a third person, for

example a colleague or the secretary. Additionally, the user wants to be aware of the calls, but does not want to be disturbed. Therefore, the phone goes into QUIET (Q) mode, where no ringtone or vibration is used, a light is used to signal the call. However, the quiet mode is only used if the environment noise is not too high. Suppose the user enters a meeting, in which there is a perceivable high level of noise. As the user is entering a meeting, the QUIET contexts should be activated (as a forwarding request from the activation of the MEETING context). However, a NOISY context is active (due to the sensed noise level) impeding the QUITE context from becoming active. The interaction between these three contexts is shown in Figure 9.7. Using the suggestion dependency relation, instead of requesting the activation of the context QUIET, the context MEETING merely suggests its activation. As QUIET cannot be activated, the behavior of the application would be a call forwarding, while playing the user's ringtone, which is a more adequate behavior given the surrounding execution environment.

The activation semantics of this scenario is as follows: When the activation for the meeting context is requested, transition req(M) is fired. At this point the only enabled transition is act(M), which fires adding a token to context place M and temporary place Pr(Q) (the state shown in Figure 9.7). At this point the enabled transitions are act(Q), and cl(Pr(Q)). However, since act(Q) has a higher priority, it is the fired transition. Firing the act(Q) transition adds a token to context place Q, and disables the cl(Pr(Q)) transition. Since there are no more internal transitions to be fired, the CoPN reaches a consistent state.

Suppose now that the NOISY context is already active in Figure 9.7. In this setting after firing the act(M) internal transition, the only enabled internal transition is the clearing transition cl(Pr(Q)), act(Q) is not active due to the inhibitor arc from N. This transition must fire, and its firing yields a state in which the only marked places are M and N, which is a consistent state.

9.3.3 Composition and Correctness Results

Now that we have introduced two new context dependency relations we must revisit Theorem 6.1 to ensure that the introduced relations can be effectively composed with the previously defined ones (Definitions 6.13 through 6.17). The two context dependency relations must also ensure that the intended semantics of the relation is preserved for any kind of CoPN.

Extended result of Theorem 6.1

Proof. Again, let us suppose a context dependency relation $\langle R, \mathcal{C}_{\mathtt{A}_1}, \ldots, \mathcal{C}_{\mathtt{A}_n}, \mathcal{C}_{\mathtt{B}} \rangle$ with associated functions $\mathtt{ext}_R$ and $\mathtt{cons}_R$, where the singleton CoPNs $\mathcal{C}_{\mathtt{A}_j}$ are the source contexts of the dependency relation for $1 \leq j \leq n$, and $\mathcal{C}_{\mathtt{B}}$ is the target singleton CoPN of the dependency relation.

First, note that the all enabling conditions defined in Equations (9.1) through (9.7) are covered by the two enabling conditions $\mathtt{A}_j \in t\bullet \wedge \mathtt{A}_j \notin \bullet t$ or $\mathtt{A}_j \in \bullet t \wedge \mathtt{A}_j \notin t\bullet \wedge \mathtt{B} \notin \circ t$. Consequently, we must prove that none of the new arcs interferes with such conditions. Secondly, note that Equations (9.4) through (9.7) correspond exactly to Equations (6.4) through (6.7). Since the arcs added by the suggestion dependency relation do not invalidate any of the enabling conditions, and thus the $\mathtt{cons}_S$ function can be applied in any order with all other $\mathtt{cons}_R$ functions.

$\mathtt{A}_j \in t\bullet \wedge \mathtt{A}_j \notin \bullet t$: Equation (9.1) introduces an arc from the activation transitions of the source context places to the preparing to activate temporary place introduced in function $\mathtt{ext}_\vee$. Similarly, Equation (9.2) introduces an arc from the deactivation transitions of the source context places to the preparing to deactivate temporary place introduced in function $\mathtt{ext}_\vee$. None of these equations invalidate the enabling condition of other context dependency relations. Hence, the $\mathtt{cons}_\vee$ function can be applied in any order with all other $\mathtt{cons}_R$ functions.

$\mathtt{A}_j \in \bullet t \wedge \mathtt{A}_j \notin t\bullet \wedge \mathtt{B} \notin \circ t$: In the case of this enabling condition the reasoning is similar to that given for the previous case, since the $\mathtt{cons}_\vee$ only adds arcs to temporary places, the enabling conditions of the transitions are not invalidated. Consequently, the $\mathtt{cons}_\vee$ function can be applied in any order with all other $\mathtt{cons}_R$ functions.

Since none of the newly introduced arcs interfere with the enabling rules, we can assert that the $\mathtt{cons}_R$ functions can be applied in any order for the seven context dependency relation types $\{E, C, I, Q, \wedge, \vee, S\}$. □

Correctness We now illustrate how the intended behavior of the new context dependency relations is preserved when these context dependency relations are composed into general CoPNs.

Disjunction dependency relations add places and transitions by means of the $\mathtt{ext}_\vee$ function to mange the activation state of all the contexts involved in the dependency relation. The state of all contexts is represented by a new context place, labeled $A_1|\cdots|A_n$. Tokens are added to the new context place whenever one of the source contexts becomes active. This situation is ensured by adding an arc from *every* transition activating the source contexts, to a newly introduce preparing to activate temporary place (managing the activation of the new context place). This behavior is ensured because these arcs are added by the $\mathtt{cons}_\vee$ function after all transitions have been introduced in the CoPN. Similarly, the new context place should not be marked if none of the source context involved in the context dependency relation are active. This situation is managed by requesting the deactivation of the new context place every time one of the source contexts gets deactivated by adding an arc form *every* deactivation transition of the source contexts to the new preparing to deactivate temporary place introduced by the $\mathtt{ext}_\vee$ function.

The arcs, places, and transitions introduced by the $\mathtt{ext}_\vee$ and $\mathtt{cons}_\vee$ functions ensure that new context place is marked as long, and as many times, as the source contexts involved in the dependency relation are marked.

Suggestion dependency relations have the same structure of causality dependency relations, which means that the dependency relation behaves correctly when composed in a general CoPN as it was illustrated in Section 6.2.2. It is only left to check that the source context of a suggestion dependency relation can still be activated in the case the target context cannot be activated. This case is managed by the clearing transition introduced in the $\mathtt{ext}_S$ function. For *every* activation transition of the source context, the activation of the source context is requested. However, if the later activation cannot be processed (e.g., because the activation transition of the target context is not enabled) then the clearing transition becomes enabled and fires. This avoids reaching an inconsistent state

of the CoPN, thus, the source context can be effectively activated even if the target context cannot be activated.

9.3.4 Evaluation

To evaluate the extensibility of CoPNs we show how the definition of context dependency relations can incorporate new interactions between contexts. The introduction of new context dependency relations does not require us to modify the consistency verification of the system, new context dependency relations are verified in the same way as those already defined in the system. This section introduces two new context dependency relations, namely disjunction and suggestion.

We first introduced the disjunction dependency relation. As claimed in Section 6.2.2, the introduction of new context dependency relations only requires us to express the constraints representing the interaction between the contexts, that the CoPN must satisfy. This was illustrated by the introduction of the disjunction dependency relation, which only requires the definition of four dependency constraints to introduce an new kind of interaction between an arbitrary number of contexts. Note that the introduction of the disjunction dependency constraint, as it was the case of the conjunction dependency constraint, defines a context that cannot be activated or deactivated directly, but rather it is activated according to the conditions of other defined contexts. Nonetheless, this seemly new fundamental way of context activation does not require any modification to the activation semantics of CoPN. Similar approaches as *composite layers* do require to modify the formal basis behind the activation of contexts to account for these kinds of interactions between contexts [105].

Introduction of the suggestion dependency relation, however, required more of work. This is due to the fact that the firing semantics CoPNs do not manage conditional firing of transitions. The suggestion dependency relation requires us to test for the possibility of activating a context. The outcome of such a test differs according to whether the conditions of the test are met or not. In order to account for such a decision process, we added (conceptually) a new element to the CoPN model, namely clearing transitions. However, the firing semantics of the model's execution were not changed. Clearing transitions keep the semantics of the system by taking advantage of the firing facilities provided in the definition of static transition priorities. The conditional flow of tokens is managed by transitions of different priorities.

9.4 Context Petri Nets Performance Evaluation

To round up the evaluation of CoPNs, we conduct in this section a performance analysis of our run-time model. As mentioned previously, the objective of CoPNs is to provide a sound basis for the development of Dynamically Adaptive Software Systems. The developed basis in this dissertation presents a run-time model to manage the consistent activation and deactivation of adaptations during program execution. Since we are reasoning dynamically about the state of adaptations as they become active and inactive, it is expected we observe a decrease in the overall performance of systems at run time. In this section we evaluate the performance of the CoPN run-time model with respect to the performance of the execution of Subjective-C. The purpose of this evaluation is to verify if the CoPN model can be effectively used for a

running application, or whether the performance overhead is so large that it renders Dynamically Adaptive Software Systems unusable.

The benchmarks were run on an Apple MacBook Pro with 2.53GHz Intel Core 2 Duo processor and 4GB of RAM running OSX version 10.7.4 and the LLVM 3.0 implementation of Objective-C 2.0.

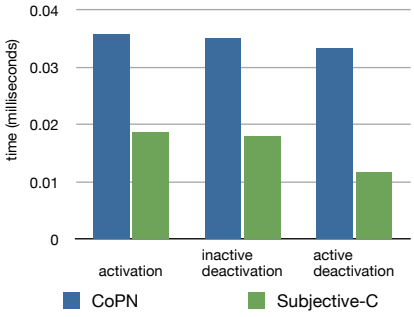
Our analysis is based on the activation and deactivation of a context in different situations with respect to its activation state and context dependency relations. Six different benchmarks were run. Each benchmark was run 5 times and in each benchmark a context was activated and deactivated 1000 times. Figure 9.8 shows the results of these benchmarks, counting the average time (in milliseconds) for the (de)activations.

Figure 9.8 shows the different executed performance analyses. Each of the graphs shows a time in milliseconds (y-axis) vs. size of contexts defined in the system (x-axis). Figure 9.8a shows, respectively, the activation of a context, the deactivation of an inactive context, and the deactivation of an active context, for CoPNs (left column) and Subjective-C (right column). Figure 9.8b shows the time to activate and deactivate a context at the beginning of a chain of implication dependency relations, depending on the length of that chain. Figure 9.8c shows the time to activate and deactivate a a context belonging to a graph of contexts, where every context is in an exclusion dependency relationship with the other contexts in that graph. Figure 9.8d shows the time to activate and deactivate a context in a cycle of exclusion dependency relations, where each context excludes the following and the last context excludes the first. Figure 9.8e shows the time to activate and deactivate a context that has an exclusion dependency relation with all other defined contexts. Figure 9.8f shows the time to activate and deactivate a context that has an implication dependency relation with all other defined contexts . Figures 9.8b to 9.8f show the activation/deactivation tests for a varying number of contexts, from 5 to 50.

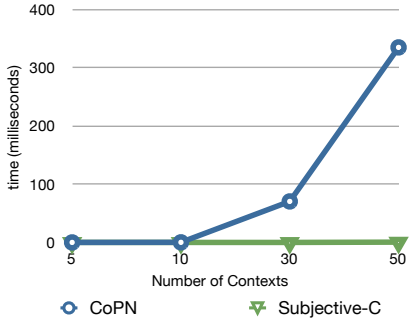
Although there is a (sometimes considerable) overhead for the activation of contexts using CoPN, the overhead was expected due to the fact that CoPN executes additional checks dynamically to ensure the system's consistency. Moreover, the increase in the execution time as the number of contexts increases can be explained by the way in which context activation requests propagate through the CoPN. A request must be issued for every context. Thus, the time to activate a context increases as the number of contexts increases.

For a small number of contexts (between 5 and 10) CoPNs's execution time remains comparable to that of Subjective-C. Regardless of the number of contexts defined and the types of context dependency relations defined among them, in the worst case scenario the execution time remained under half a second, which is still acceptable. Note that the worst performing cases (a chain of implication dependency relations between 50 contexts, and a complete graph of 50 contexts of exclusion dependency relations) are unlikely in a real world application. Even more, remember from Sections 9.1 and 9.2 that we reused two applications previously developed with Subjective-C, in the setting of CoPNs. In both cases, there was no perceivable difference in the user's experience when running either the Subjective-C version or the CoPNs version.

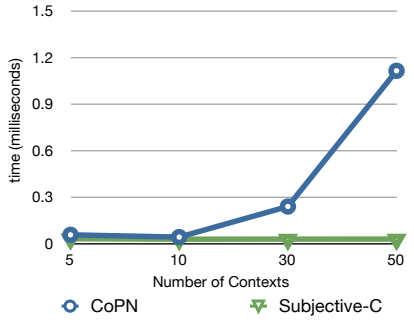
The current implementation of CoPNs is optimized. However, optimization to increase its performance could be envisioned. Section 11.4.4 discusses some possibilities that can be implemented in CoPNs to improve its run-time performance.



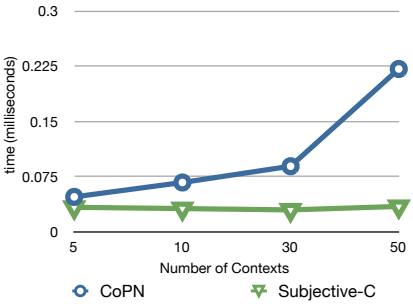
(a) Single context (de)activation.



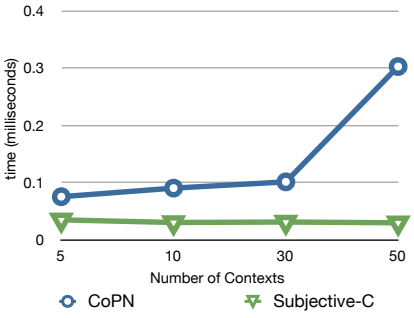
(b) Chain of implications graph.



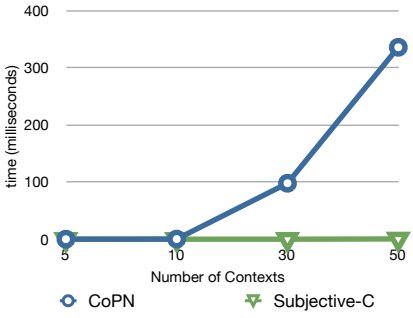
(c) Complete exclusion graph.



(d) Complete exclusion cycle.



(e) One to many exclusions.



(f) One to many implications.

Figure 9.8: Benchmark results for the activation and deactivation of contexts with context dependency relations in CoPNs and Subjective-C.

9.5 Conclusion

This chapter evaluates CoPNs with respect to three different view points here, its *usefulness*, *appropriateness*, and *extensibility*. We recapitulate the most important points demonstrating the benefits of having a theory and sound programming model for the development of Dynamically Adaptive Software Systems.

First, the formal basis introduced with CoPNs is useful during the development process of Dynamically Adaptive Software Systems because it allows us to reason about the definition and interaction between adaptations. The reasoning power of the CoPN model can be used to identify incoherences and behavioral inconsistencies in the system, Moreover, it can be used to improve the adaptability of the system, as Section 9.1 showed. CoPNs also prove to be useful for language designers. The formal definition of context dependency relations and the reasoning power provided by CoPNs can help to spot inconsistencies in the language. This was the case in Subjective-C for the definition of the implication dependency relation, as mentioned in Section 7.1. Additionally, the support provided in CoPNs by means of the context activation simulator proved useful to spot another behavioral inconsistency in the case of the requirement dependency relation, as shown in Section 8.2.

Second, the formal basis introduced with CoPNs proved appropriate for the definition and modeling of Dynamically Adaptive Software Systems by allowing the use of the model for the formalization of new scoping mechanisms for adaptations, by means of colored CoPNs. Colored CoPNs enabled us to cleanly separate local and global adaptations, while allowing the interaction between adaptations activated locally and those activated globally, as shown in Section 9.2. The requirement of separating local and global adaptation is proposed by different types of Dynamically Adaptive Software Systems, broadening the scope of applicability of our theory. Even more, this extension opens the door to study concepts of distribution of Dynamically Adaptive Software Systems from within the same formal framework used in the specification of the rest of the systems.

Third, the formal basis introduced with CoPNs is easily extensible to cover new capabilities and requirements of Dynamically Adaptive Software Systems without the necessity of modifying the semantics of the system execution. The programming model of CoPNs is developed as general as possible in order to allow its extensibility in the future. Section 9.3 shows part of the extensibility of the programming model by using the definition of static transition priorities to introduce the suggestion dependency relation. However, other aspects of the formal basis can be used to extend the computational model of CoPNs, covering different types of Dynamically Adaptive Software Systems. For example, the definition of the flow function can be used to allow the flow of multiple tokens through arcs, or the marking multiset is used to integrate colored Petri nets into our programming model.

All along this chapter, we have shown how to use and extend CoPNs to consider other aspects of Dynamically Adaptive Software Systems. CoPNs can easily be extended by introducing colors (marking multi-sets), new context dependency relations, or using the multi-flow arcs (multiple tokens flow through an arc). We can assert that the programming model presented in this dissertation effectively provides a basis for the specification, management, and development of more general types of Dynamically Adaptive Software Systems.

Chapter 10

Putting Context Petri Nets in Perspective

The formal basis and programming model developed around CoPNs has proven useful for the formalization, representation, run-time consistency management, and analysis of context-aware systems as realized by COP languages. However, the formal basis of CoPN presented in this dissertation is not restricted to COP systems. The results and developments of our approach can be translated to the broader class of Dynamically Adaptive Software Systems, or even other systems in which similar adaptation processes are used.

This chapter evaluates the appropriateness of the formal basis proposed by CoPNs by revisiting some of the approaches realizing Dynamically Adaptive Software Systems, discussed in Section 3.1. In the following, we put CoPNs in perspective of two architectural approaches, two middleware approaches, and one programming language approach realizing DASS. For each of the approaches, we describe how the CoPNs model complies with their requirements, and whether CoPNs could assist in tackling existing challenges for these application domains.

10.1 Dynamic Software Upgrades

As discussed in Section 3.1.1, dynamic software upgrades are used for the introduction of new functionalities or the correction of bugs in a running software system. Section 3.1.1 presents the four basic properties that dynamic software upgrade systems should provide: *timeliness*, *flexibility*, *robustness*, and *practicality*.

This section discusses how the formal basis proposed by CoPNs complies with the requirements of dynamic software upgrades. CoPN exposes a transparent mechanism for the definition, inclusion, suppression, and composition of behavioral adaptations. Since context activation occurs at run time, the CoPN model can be mapped directly to the computational model of dynamic software upgrades.

The support for managing the dynamic introduction and withdrawal of adaptations to a software system presented by CoPNs can also be used to support dynamic software upgrades of a system under the following assumptions: (1) contexts are mapped to software upgrades, where each context represents a system version, (2) the behavioral adaptations associated with a context are mapped to the behavior introduced by each upgrade. (3) context activation corresponds to the deployment of a software upgrade, and (4) context composition is the composition of upgrades to be deployed with the

upgrades already deployed in the system.

CoPNs also provide support for the deactivation of contexts, and thus the recomposition of the system due to missing behavior. Such a property could be mapped to system downgrades. However, most dynamic software upgrade systems do not provide support for downgrading the system. Later in this section we discuss this and other additional capabilities provided by CoPNs for the development of dynamic software upgrades.

Timeliness stands at the core of Dynamically Adaptive Software Systems, and in particular CoPNs. Behavioral adaptations are made available whenever their associated context is rendered active according to the surrounding execution environment of the system. As behavioral adaptations are included to and withdrawn from the system, it can continue working without any disruption; the adapted behavior is applied on the fly as required. However, in some dynamic software upgrade systems it is required to wait for a *quiescent* state [114] in order to apply any upgrade —that is, a state in which the behavior to be adapted is not in use. The introduction of quiescent states is motivated as a means to manage/avoid inconsistencies between the running behavior and that introduced by software upgrades. As a consequence, it might be that systems must wait long periods of time until reaching a quiescent state. Even worse, such a state is not guaranteed ever to be reached. The approach for the introduction of software upgrades undertaken by CoPNs does not require the system to reach a quiescent state, while still ensuring the consistency of the system's behavior after the upgrade has been introduced. System consistency is enabled in CoPNs by means of context dependency relations an the run-time management of context activations.

Flexibility In dynamic software upgrade systems, upgrades take place at one of two granularity levels. Fine-grained upgrades (e.g., replacing methods or fields) might be too detailed, presenting the difficulty that some elements to be upgraded are tightly coupled with other system elements. In such case, ensuring that the upgrade is safe with respect to the system state can be quite challenging. Before upgrading an element it must be ensured that all other entities that use or are used by such an element are not active at the moment of the upgrade, and that they comply with the upgraded element. Coarse-grained upgrades (e.g., replacing full components) present a clean way to isolate parts of the system state and hence ensure safety through, for example, waiting for transactions between components to reach *quiescence* [114].

COP systems, and hence CoPNs, allow us to define behavioral adaptations at different levels of granularity. Behavioral adaptations associated with a context may define adaptations for methods, variables, or complete classes. This means that through CoPNs, it is possible to extend the *flexibility* with which software upgrades are defined. The main motivation of dynamic software upgrades to support coarse-grained upgrades (upgrading application components), is to preserve the consistency of system behavior. CoPNs guarantee a consistent behavior of the system in the light of dynamic adaptation regardless of the type of behavioral adaptions associated with the context. Thus, finer-grained software upgrades can be introduced by means of CoPNs without compromising the consistency of the system.

Robustness Ensuring the robustness of the system when dynamically deploying software upgrades is a difficult problem. Different dynamic upgrade systems [114,

187] require programmers to manually identify and define safe points in the system for upgrades to take place —that is, points in the system execution for which introduction of upgrades does not break behavior. For example, inconsistencies may arise by accessing an instance variable that was changed or no longer exists after the upgrade, or by modifying the interface provided by a service. The robustness property presents a trade-off with the timeliness one, as the system must wait until the defined safe point is reached in order to deploy the upgrade. The approach of statically defining the safe points at which the system can process upgrades may become cumbersome or even unfeasible when upgrades are unannounced, as it occurs for example in COP systems. Even more, CoPNs allow a fine-grained definition of software upgrades which could make the definition of safe system states problematic due to their multiplicity. The approach taken by CoPNs for ensuring the robustness of dynamic software upgrades is twofold. On the one hand, CoPNs offer the possibility to reason about system properties and identify incoherent upgrade definitions, and possible behavior inconsistencies that upgrades might introduce. On the other hand, as mentioned previously, the run-time management offered in CoPNs ensures that the introduction of software upgrades is not inconsistent with other upgrades (currently) deployed in the system.

We argue that the existing trade-off between timeliness and robustness disappears with CoPNs. Software upgrades that do not break the system are deployed immediately. Software upgrades that may break the the system's behavior are refused —that is, they are not deployed. The decision of disallowing potentially harmful upgrades does not contradict the timeliness property, because if an upgrade might break the behavior of the system, it is not the most appropriate behavior according to the surrounding execution environment.

Practicality Upgrade definitions and their dynamic deployment in the system are realized by specific language constructs in CoPNs. Using such constructs, the definition of software upgrades becomes a natural part of the system's structure. Deployment of upgrades is performed automatically in CoPNs as a reaction to changes in the surrounding execution environment of the system. CoPNs offer additional support for the deployment of software upgrades. For example, whenever it is not possible to perform the upgrade, due to the upgrade being identified as harmful, CoPNs signal the error and the cause of that error to the user. We argue that these properties facilitate the realization of dynamic software upgrades, rendering them practical and transparent to programmers using CoPNs.

In addition to the aforementioned basic requirements of dynamic software upgrades, CoPNs also provide support for other characteristics not present in standard dynamic software upgrade approaches. We discuss these characteristics here.

Safe upgrades: As discussed previously, the focus of CoPNs is on ensuring the consistent execution of a software system as it is upgraded dynamically. To avoid upgrades that are not safe with respect to the current behavior provided by the system, and the upgrades currently deployed, CoPNs undertake two approaches. First, during the system's development it is possible to reason about system upgrades as they are defined. Such a reasoning process allows avoiding incoherences and inconsistencies in the definition of upgrades (Chapter 7). Second, a run-time verification process is implemented to manage the dynamics of upgrades as they are deployed in the system. This verification process allows us to

protect the system from hazardous upgrades that may be introduced dynamically (Chapter 6).

Version Loyalty: Version loyalty refers to how faithful the execution of an application is, with respect to a particular configuration of software upgrades. CoPNs can enable version loyalty by means of upgrade independence —that is, the possibility to isolate two different configurations of deployed upgrades in an application. Two different approaches can be taken to enable version loyalty. The first approach, consists of using the capabilities of the COP language to isolate the state of upgrades. State and behavioral adaptation definitions are confined to a particular upgrade, and are made available whenever the upgrades are deployed in the system [44, 11]. A second approach, could be conceived by further extending the execution model of CoPNs, as done in Section 9.2. Colored CoPNs were introduced with the objective of isolating deployed upgrades to a particular execution thread. This same technique can be used to provide version loyalty. Each upgrade is deployed with a particular token color, marking the adaptation as active for said color. Behavior and state associated with the upgrade are only accessible for the application executions (e.g., threads) associated with that color. Independence of software upgrades would allow us to provide multiple running versions of a system. For example, by enabling specific features in a program according to different user profiles.

Consistent Downgrades: CoPNs allow us to dynamically withdraw behavioral adaptations from the system. In the context of dynamic software upgrades, this property could be used to allow system downgrades. Normally, dynamic software upgrades systems do not provide support for downgrading the system. However, we recognize the usefulness of supporting this property, for example, in case an upgrade introduces erroneous behavior, or to introduce new upgrades that might be in conflict with currently deployed ones. CoPNs allow the withdrawal of any upgrade in the system, ensuring that the perceivable behavior of the system is not compromised. Removing upgrades is ensured not to lead to inconsistencies using the run-time consistency management provided by CoPNs. Behavioral adaptations are automatically reordered according to the deployed upgrades (Section 6.4.2), and resolved using such ordering and the method reuse mechanism of COP systems (Section 4.1.2).

10.2 Dependency Injection

COP was first introduced as an alternative approach to design patterns for the dynamic adaptation of behavior in software systems. As is, COP already provides an alternative implementation to the introduction of behavioral adaptations using design patterns. On top of this, CoPNs offer run-time consistency management capabilities that COP languages do not. As an example of how different design patterns for dynamic behavior adaptation could be replaced by COP, and in particular by CoPNs, we examine the dependency injection pattern. Dependency injection was discussed in Section 3.1.2 as a middleware approach that allows the introduction of behavior at specified points of a running system.

Dependency injection is normally implemented in software systems that require the management of different variations of the system, while maintaining one instance of it. Three important requirements for customizing software systems are identified,

with respect to application development, application configuration, and run-time support [184].

Software adaptation definition: Programmers of the software system should be offered a simple way to manage the different specified adaptations, as separate units of deployment that can be selectively bound to the core architecture of the system.

Configuration facility: With respect to customization, system users should be offered a configuration facility to select which adaptations should be enabled for them. In addition, this facility should also allow them to specify particular configuration parameters, such as, business rules or user's preferences. The independence of these configuration data should remain isolated for each specific system adaptations.

Run-time activation of software adaptations: Run-time support is needed to provide support for activating adaptations on an isolated basis, for example, per user. Whenever a user starts the system, the situations relevant to the user's surrounding execution environment should be determined. Based on the acquired information, run-time support should then activate the appropriate behavioral adaptations to process the requests of the user. Another key requirement of run-time support is that behavioral adaptations should be applied in an isolated way without affecting the behavior of services that is provided to other adaptations.

We now discuss how the different programming facilities provided by COP, and in particular CoPN, can be used to address these requirements extracted from Dynamically Adaptive Software Systems using dependency injection.

Software adaptation definition: COP allows the definition of behavioral adaptations and their association with particular situations in the surrounding execution environment of the system, as explained in Section 4.1.1. Such definitions ensure that behavioral adaptations are grouped by their context, where contexts are independent from each other. Additionally, the internals of COP automatically manage the integration of behavioral adaptations into the base system without requiring any modification to the system's architecture, as it is the case with most design patterns, and dependency injection in particular.

Configuration facility: The activation model of behavioral adaptations, described in Section 4.1.2, allows us to activate adaptations as a reaction to users' input. For example, the `@activate` construct of CoPNs can be used to activate a context and make available its associated behavioral adaptations when users select their preferences in the system. Independence of behavioral adaptations is accomplished programmatically by associating a particular state of the system with specific adaptations. Such a state is only accessible whenever the context with which it is associated becomes active.

Run-time activation of software adaptations: The computation model of COP systems allows us to gather information about the system and automatically activate the corresponding adaptations, for instance as it is done by the context gathering module in CoPNs (Section 8.1). Additionally, COP systems offer different ways in which adaptations may interact, as explained in Section 4.1.4. CoPNs ensure, for example, that interaction between adaptations remain consistent —that is, activation of adaptations does not yield contradicting system states with respect to other active adaptations.

A performance evaluation comparing the dependency injection implementation of the web booking application (Section 2.3.2) with its COP counterpart is provided by Truyen et al. [184]. The performance evaluation proves the usefulness of using COP as an alternative to architectural solutions for implementing Dynamically Adaptive Software Systems.

10.3 Self-Adaptive Systems

As discussed in Section 3.1.2, self-adaptive systems tackle the problem of costly re-configuration and re-deployment of software systems by the introduction of *dynamic adaptations* —that is, system re-configuration without requiring any downtime. In order to allow dynamic adaptations, self-adaptive systems are required to *monitor*, *analyze*, *plan*, and *execute* both the internal execution of the system and its surrounding environment. To support these requirements a system must fulfill a set of characteristics, called self-* properties [159].

Different self-* properties cover different aspects of dynamic adaptivity, ranging from acquiring information and self-awareness of the system's surroundings, to decision making and self-adaptiveness. Multiple self-* properties can be fulfilled in a system, based on the objective of the system and its quality requirements. However, it is common practice for self-adaptive systems to be developed with one particular property in mind. As a consequence, there is no single programming model that covers all the existing self-* properties. Furthermore, creation of self-adaptive systems can be cumbersome. A series of challenges must be addressed to provide fully self-adaptive systems. Here, we discuss some of the existing challenges [118, 159].

C.1 The first challenge evidenced in the development of self-adaptive systems, is that they often do not provide support for all self-* properties. The majority of self-adaptive systems focus on one particular self-* property. Hence, interaction between properties is not taken into account. A unified coordination model for all self-* properties is required.

C.2 An important challenge is highlighted in the system's ability to detect evolving situations of its execution environment and consequently present appropriate adaptations to them. Questions that need to be tackled include: *how are changes propagated and adapted in the system model? How to isolate particular changes that do not concern the totality of the system? How to detect adaptations that may harm the system's behavior?*

C.3 The process in which selected adaptations are composed with the base system is still an open question in self-adaptive systems. The process deciding whether an adaptation should be included in the system, has historically been a static process. Even in the static case, there is no consensus yet about how to approach situations in which there is incomplete information about the system and its adaptations (e.g., in the case of decentralized system where parts of the system constraining the composition of new adaptations may be temporarily offline), or the interaction between global and local decisions to compose adaptations.

C.4 A challenge related to the previous one, is that of how to ensure that introducing a new adaptation yields the desired system behavior.

C.5 Adaptations interact through adaptation policies or rules. The problem with this interaction is that such rules are typically expressed in a high-level lan-

guage which often differs from the one in which the system is implemented. The translation process usually loses the intended semantics given by the high-level definition, because this semantics is not easily expressible in the system or are defined in an imprecise way. The implementation of the policy rules can put in evidence errors in the interaction defined between adaptations. Additionally, interaction rules are often hard-coded, limiting the system to a single set of possible adaptations.

C.6 Testing remains an open issue in engineering self-adaptive systems. Due to the multiplicity of applicable adaptations, multiple execution scenarios can branch at every program point. Initial test mechanisms and model checking-based verification techniques have been proposed to cover testing of self-adaptive systems.

In the remainder of this section, we explore how the different language facilities and tools developed in the CoPN programming model could be used to provide support for self-* properties and some of the challenges in self-adaptive systems. Similar studies have recently been developed providing support for self-adaptiveness by means of COP languages [161].

Providing support for self-* properties with CoPN

Given the close relation between COP and self-adaptive systems (Table 3.1), COP provides support for many self-* properties. The COP paradigm is thus a good candidate for effectively unifying the programming model realizing self-adaptive systems. The self-* properties that can be addressed using the formal basis of CoPNs developed in this dissertation include:

The **self-awareness** and **context-awareness** properties are directly supported following Definition 4.2 of contexts. In CoPN these properties are respectively supported by the *internal* and *external* information inputs to the context management module in the architecture of context-aware systems, shown in Figure 8.1.

The **self-control** property is naturally enabled through the dynamic activation of contexts as explained in Section 4.1.2. For example, using the `@activate` and `@deactivate` language constructs in CoPN.

The **self-configuring** and **self-recovering** properties are supported by means of the interaction between behavioral adaptations. In these cases, the support provided by CoPNs comprehends the use of disambiguation techniques, as explained in Section 4.1.4, the `@resend` language construct, and the context dependency relations, defined by CoPNs in Section 6.2.2.

The **self-organizing** property is supported by the combination of: (1) the definition of context dependency relations and the composition operator of CoPNs (Section 6.2), and (2) the introduction of the context acquisition module in CoPN, as explained in Section 8.1.

The **self-diagnosing** property is partially supported by the introduction of the analysis engine used in CoPNs (Chapter 7). Reasoning about the properties that the system should satisfy, allows us to diagnose erroneous configurations between adaptations, as explained in Section 7.1.2. The property is not fully supported in CoPNs as the analysis of the system is not fully automated. Section 11.4 discusses different ways in which the analysis of the system could be fully automated and extended to support the self-diagnosing property.

The **self-healing** and **self-protecting** properties are partially supported in CoPN by the run-time verification of consistency between adaptations. Section 6.3.1 explains how CoPNs can recover from errors before they are actually woven into the system. However, the CoPN model only supports one corrective action, that is, rolling back to the last correct state. Other possibilities for providing self-healing with CoPNs are discussed in Section 11.4.3.

Tackling challenges of self-adaptive systems with CoPN

The formal basis and programming model developed in CoPNs exhibit most of the characteristics expected from self-adaptive systems. We now turn our attention to the previously enumerated challenges in the development of self-adaptive systems, and observe if they can be addressed using CoPNs.

Challenge C.1 is addressed by CoPNs as a whole. As it is possible to see from the preceding overview of self-* properties, the development of the CoPN formal basis satisfies the requirements for a sound and unified coordination model of self-adaptive systems.

Challenge C.2 is addressed by different parts of the CoPN model. The context gathering module introduced by CoPNs is in charge of detecting changes to the surrounding execution environment, and forwarding the respective events to the CoPN representation of the system (as explained in Section 8.1). Context activations are propagated by means of context dependency relations within the CoPN (as given in Sections 6.2.2 and 6.3). Isolation of adaptations not concerned with the complete system is managed using global and local context activations (as explained in Section 9.2). Finally, harmful situations to the system are addressed by the execution semantics of the system (as explained in Section 6.3.1).

Challenge C.3 is partially addressed by the dynamic disambiguation characteristics offered by CoPN for the selection of behavioral adaptations. Additionally, CoPN offers a unique interaction model between local and global behavior (as explained in Section 9.2). The CoPN programming model assumes a complete information world —that is, a system where all the information about contexts and their associated behavioral adaptations is known. Nonetheless, modifications and customization of the analysis processes provided with CoPNs could be applied in order to allow the partial verification and analysis of the system —that is, customizing the analysis algorithms so that only part of the CoPN is reasoned about.

Challenge C.4 is addressed by the introduction of context dependency relations (Section 6.2.2), and their run-time verification of the consistency of the system's CoPNs (Section 6.3.1).

Challenge C.5 is addressed by the low-level definition of contexts and context dependency relations given in CoPN (Definitions 6.2 and 6.13 through 6.17). Using the first-hand view representation of Petri nets, context dependency relations definitions are directly mapped onto their run-time representation. Moreover, as specified in Section 8.1, the context acquisition model introduced by CoPNs allows us to introduce new contexts at run time.

Challenge C.6 is addressed by the simulation tool proposed by CoPN, as shown in Section 8.2. This tool allows us to simulate the activation of a particular context

without requiring a complete running version of the system. Together with the analysis and verification steps introduced by CoPNs, the simulation tool constitutes a first step towards a framework for the analysis and testing of self-adaptive systems.

Because of the support it provides in the development of self-adaptive systems, the formal basis for COP systems presented in this dissertation by means of context Petri nets, seems to be a promising approach for the development of self-adaptive systems.

10.4 Event Systems

As discussed in Section 3.1.2, event systems allow the automatic interaction between system components. Event systems can be used for the representation of Dynamically Adaptive Software Systems, where particular behavior defined in the system is automatically triggered whenever a particular event associated with that behavior is triggered [121, 92]. Event systems are normally represented by automata or LTS [91] to facilitate the management of the system's control flow.

In this section we discuss the key characteristics of the CoPN model allowing the support and development of event systems that dynamically adapt their behavior to the surrounding execution environment of the system.

One important difference between CoPNs and event systems, is that CoPNs confine the system and its representation to part of the same programming model, unlike event systems which use two separate representations: a programming model of the system on one hand, and its representation as an automaton on the other hand. Having a single model representation facilitates the development and analysis of event systems. The association of particular system events with the dedicated behavior that must be computed by their triggering is directly supported by CoPNs with the definition of adaptations as first class entities (Definition 6.24). Adaptations are composed from the conditions in the surrounding execution environment they are most appropriate for (contexts), and their respective associated behavior (behavioral adaptations). These concepts can respectively be mapped to the concepts of: *event kind*, *event* and *event thread* in INI [121], and *conspecs*, *fields* and *methods* in EventJava [92].

Secondly, the triggering of behavioral adaptations follows the scheme of event systems, where an actuator loop observes a particular state variable of the system, or a property in its surrounding execution environment. Whenever some predefined constraints over such a state are identified as valid, the event is triggered and the behavior associated with the event is executed. In CoPNs, monitoring of events is delegated to the context gathering module, described in Section 8.1. Once an event is identified, its triggering is deferred to the underlying CoPN (the context management module) for inclusion of the appropriate behavior associated with the event. Behavior associated with the event is included if and only if the firing step of the CoPN is consistent and the resulting state of the system differs from the last consistent state.

Finally, representation of event systems by means of CoPNs, is an improvement with respect to their classical representation as automata. The first shortcoming identified for the representation of events by means of automata is that events are treated independently. That is, events have a one-to-one association with states of the automaton, edges in the automaton represent the actions performed by the system to go from one state to another. In order to represent multiple states taking place simultaneously in the surrounding execution environment of the system (i.e., constraints defining two

events are both satisfied) additional states need to be added to the automaton. Adding states can lead to a state space explosion in the case of large systems (Section 3.2.5). The Petri net representation of event systems allows the interaction between events by considering them as a whole in the surrounding execution environment of the system.

Additionally to the aforementioned characteristics, CoPNs offer the possibility of interactions between events and the management for such interactions. Normally, events are seen as isolated circumstances taking place in the surrounding execution environment of event systems. However, since events occur as a whole, they must be addressed as a consistent set of events, rather than in isolation. CoPNs offer the possibility to define interaction between different events throughout the definition of context dependency relations (Section 6.2.2). Moreover, CoPNs address (to some extent) the shortcomings identified by rule-based modifications of event systems [121] and the interaction between *conspects* (context aspects) [92] by ensuring the safeness of applying the behavior associated with dynamic events.

The execution semantics presented in Chapter 6 ensures the safety of triggering application behavior associated with events is the system —that is, the CoPN model allows us to manage the interaction between events consistently. Consequently, we claim that CoPNs could be used for the implementation and analysis of event systems.

10.5 Reactive Programming

As discussed in Section 3.1.3, reactive programming allows the computation of events that continuously change over time in the surrounding execution environment. Section 3.2.5 discusses dataflow graphs, which are directed graphs used, among others, for the representation of reactive programs. Dataflow graphs are implicitly generated by the reactive system (i.e., the reactive programming language). Dataflow graphs can also be used as a visual coordination model to inspect and edit the control flow of the system [129].

The computation model of dataflow graphs can be described by three main characteristics. These characteristics comply with the model of reactive programming.

States and operations: Dataflow graphs represent input and output states and the operations taking place between them. Operators and states are represented by nodes in the graph.

Node communication: Information and data about computation flows between nodes by means of *dataflow edges*. Information always flows from the output of an operator to the inputs of other operators or states.

Information transfer: Information is transferred between two nodes by one of two semantics. The first semantics consists of executing operations in an operator node whenever *all* of its inputs have received a value. The second semantics consists of executing operations in an operator node whenever *any* of its inputs have received a *new* value. Upon execution, operators take the information from all their inputs, and set their computation in their outputs.

We now describe the similarities between the programming model developed in this dissertation, and the programming model of reactive systems. Two points are worth noticing before going any further. First, the support provided for event management in CoPNs is divided in two parts, namely an external events management (provided by the context discovery module in Section 8.1), and the propagation of events internally

in the system (provided by the firing semantics of CoPNs in Section 6.3.1). As a consequence, CoPNs do not provide the same support for behaviors and values as regular reactive programming languages. Second, given the different approaches taken for the development of dataflow graphs and CoPNs, the implementation of CoPNs as dataflow graphs for the coordination of reactive systems requires to modify the CoPN's formalization. This is in contrast to the support provided by CoPNs for the other approaches discussed in this chapter.

Supporting dataflow graph computation style with CoPNs

According to the aforementioned characteristics, the programming model of CoPNs can be easily mapped to the programming model for dataflow graphs . Two important similarities between CoPNs and dataflow graphs are proposed. (1) As it is the case for dataflow graphs, generation of CoPNs is done implicitly by the programming language, and (2) CoPNs also provide a visual model representation of the system allowing its inspection and modification.

State and operations: CoPNs offer a first-hand view of the system's states and its operations, where states are represented by places and operations are represented by transitions. This is the same representation given by dataflow graphs. Nonetheless, states and operations in CoPNs are "information-less". Tokens only provide information about the state of places as discrete multisets of values, where each token in the multiset describes whether the state has the property abstracted by the token. However, tokens do not carry any additional information about the system's state variables or computations. Consequently, transitions only take tokens from their input places and generate (non-mutated) tokens for their outputs but do not perform any computations.

To better correspond to the intention of behaviors and operators from dataflow graphs, CoPNs could extend the representation of tokens by providing them with information about system properties. That is, tokens might carry specific states of the system as values, for example *"variable x has value 3"*. Transitions are consequently extended to perform operations over such token information, for example *"increment the value of x by 1"*. Transitions may modify token information and transmit this new information to their output places. Luckily, such modifications of the CoPN model does not need to be introduced from the ground-up. Petri net extensions such as algebraic Petri nets [55] or the box algebra [17] allow us to annotate transitions with process expressions that can modify the information of tokens, in the same way operations in dataflow operators modify their information inputs. Moreover, programmatically, CoPNs were conceived with the necessary hooks to extend their computation model. However, the CoPN programming model still needs to be modified to provide such support.

Node communication: The communication between nodes in a CoPN follows the same communication style as dataflow graphs. Information flows through the arcs in the CoPN by means of Petri net tokens. CoPNs impose an additional restriction, since that information may only flow from outputs of states into inputs of operators, and from outputs of operators into inputs of states. This restriction adds additional nodes, which could lead to a state explosion problem as the system grows.

Information transfer: CoPNs only provide support for one of the information transfer semantics of dataflow graphs. In CoPNs, operators are triggered only when there is new information for *all* of their input states. Whenever this is the case, operators are immediately executed (either internal or external transitions). Providing support for the other semantics for triggering operators would require us to modify the semantics of Petri nets and the formal definition of CoPNs. Hence this semantics are not part of the CoPN model. Evaluation of operators in a reactive system using CoPNs does not occur every time one of their inputs changes, rather, re-evaluation of operators takes place once all input information of the operator has change. The model does not react to small localized changes.

Additionally to the fulfillment for the basic characteristics of the dataflow graph's computation model, the CoPN model also provides support for other specific characteristics provided in particular reactive and dataflow programming languages.

CoPNs allow the adaptation of behavior at different levels of granularity. This property could be shifted to the computation of dataflow graphs which manage information transfer between components. Using an extension of CoPNs in which tokens are used as data entities of the program an carry information between places, CoPNs can be used to manage the information flow between any kind of modules in the system.

Some languages to program dataflow graphs, such as Aurora* and Medusa [36], present properties for the automatic reaction to failures. In these languages, operators keep a link with their input nodes by sending periodic messages to them. Whenever there is no response from one of the input nodes after a predetermined time has elapsed, the node is considered to have an error, and the input is sought from another node. Such automatic correction of potential erroneous states can be considered as a self-healing property of these languages. As explained in Section 10.3, CoPNs provide partial support for such corrective actions. The extension of the CoPN model to provide self-healing, discussed in Section 11.4.4, takes into account the discussion presented here.

Reactive programming presents a latent problem of *glitches*. Glitches occur whenever the computation of operations can be broken down into two or more simpler components sharing a common reactive input (i.e., a value). The computation of the whole operation can be compromised depending on the order in which the operation components are evaluated. To avoid glitches in the evaluation of multiple operators that depend on a varying value, some languages, such as FrTime [42], allow the definition of a *height* for operators, such that every operator has a larger height than all of its predecessors (i.e., inputs). Operators with a smaller height are computed before those with a larger height. This definition of heights is naturally given in the CoPN model by transition priorities. Transition priorities ensure the execution of operators to take place only after all their components have been evaluated. Currently, transition priorities are set statically in CoPNs. However, it would be possible to define dynamic transition priorities [13], following the algorithm used for reactive languages. Such a definition needs to be addressed and studied carefully. It may not be feasible to consistently define priorities for all transitions when composing CoPNs.

Reactive dynamic adaptations in CoPNs

Up to this point we focused on the description of dataflow graphs and the conception of their programming model by means of CoPNs. We now proceed with describing how support for reactive dynamic adaptations can be provided in CoPNs. Dynamic adapta-

tion of behavior in a reactive setting has been proposed in the Flute language [11]. In the Flute programming model context changes take place automatically as a reaction to changes in the surrounding execution environment. Flute offers the possibility of defining the strategy with which behavioral adaptations are introduced in the system by means of procedure *interruptions*, *resumptions*, *restarts*, *immediate inclusion*, and *deferred inclusion*.

CoPNs react to changes in the surrounding execution environment by means of the event-system associated with external transitions —that is, context activations and deactivations are triggered by the context gathering module (Section 8.1). Unlike Flute, different adaptations interact with each other in CoPNs. Through the response to a change in the surrounding execution environment, further contexts can be activated and deactivated. Such an activation is performed reactively following the activation semantics proposed by CoPN (Section 6.3.1). However, CoPNs do not provide any support for the interruption or resumption of computations provided by Flute. We recognize that the two approaches are complementary and could be coupled together in the future. For example, by associating the resumption or restart strategies with internal transitions.

The coupling of the reactive programming computational model and CoPNs would provide a programming model that allows behavioral adaptations to take place reactively, enables their interaction, and ensures their consistency with respect to the surrounding execution environment of the system.

10.6 Conclusion

This chapter puts the developments of CoPNs described in this dissertation in perspective of Dynamically Adaptive Software Systems. Our research was setup in the context of providing consistency and predictability for a software system in the presence of dynamically adaptive behavior. During the development of this dissertation, we focused our efforts on one particular kind of such systems, namely Context-Oriented Programming. COP allows the implementation of software systems with highly dynamic characteristics. However, we claim that the efforts developed in this dissertation are not only valid for COP systems, but that they can actually be used in the broader context of Dynamically Adaptive Software Systems.

Throughout this chapter we validated this claim by revisiting some of the approaches for implementing Dynamically Adaptive Software Systems described in Chapter 3. Given the broad scope of Dynamically Adaptive Software Systems we did not discuss all of the surveyed approaches for the implementation of such systems in this chapter. Rather, we selected different approaches for each of the categories describing the landscape of Dynamically Adaptive Software Systems, architectural solutions, middleware solutions, and language solutions.

With respect to the architectural solutions we studied the appropriateness of CoPNs as support for the implementation of dynamic software upgrade systems and dependency injection frameworks. In both situations CoPNs effectively satisfy the basic requirements presented in each of the solutions. In addition, CoPNs provide support to facilitate the implementation process of such systems, as well as the capabilities to ensure safety properties for them. For example, ensuring safe upgrades in the case of dynamic software upgrades, or managing interaction between behavioral adaptations in the case of dependency injection. This evaluation demonstrates the potential

usefulness of CoPNs to reduce the architectural complexity of developing consistent Dynamically Adaptive Software Systems.

With respect to the middleware solutions we studied the appropriateness of CoPNs as support for the implementation of self-adaptive systems and event systems. In the case of event systems we described how the CoPN model unifies the implementation of event systems with their abstract representation. Moreover, we used the CoPN representation of event systems to treat events as a whole, rather than as isolated events taking place in the surrounding execution environment of the system. Self-adaptive systems are closely related to the purpose of COP. We showed how CoPNs adhere to the main properties of self-adaptive systems. Moreover, using the CoPN model we revisited some of the existing challenges in self-adaptive systems and studied how these can be addressed using the support provided by CoPN. This evaluation shows how the CoPN model could be useful in the case of middleware for dynamic adaptive systems.

With respect to the language solutions we studied the appropriateness of CoPNs as support for the implementation of reactive systems. Similarly to the case of event systems, CoPNs are used to unite the development of reactive systems with their representation. CoPNs can be extended to support the full vision of reactive programming as dataflow graphs. Additionally, CoPNs can be combined with existing reactive models for the adaptation of systems behavior with the purpose of enabling a consistent interaction between such adaptations. In addition to reactive programming, there is another language solution which we have not treated in this chapter, namely AOP. COP is closely related to the programming model of dynamic AOP with the difference that behavioral adaptations in COP can be introduced and customized independently according to the surrounding execution environment of the system, In dynamic AOP, on the other hand, the aspects woven in the system are necessarily known beforehand, so that their matching joinpoints can be specified. Given the similarity between the two models, the benefits that CoPNs provide to COP systems can be directly mapped to the context of dynamic AOP systems.

Given the usefulness and appropriateness of CoPNs across the landscape of different solutions for the implementation of Dynamically Adaptive Software Systems, we claim that the formal basis and programming model developed in this dissertation covers the broader scope of Dynamically Adaptive Software Systems.

Chapter 11

Conclusions

The work developed in this dissertation was motivated by the observation that, in systems allowing dynamic adaptation of their behavior, it is difficult to guarantee that the intended behavior is consistent with the actual behavior observed at run time. This problem is mainly due to the dynamic introduction and withdrawal of multiple adaptations, which may cause accidental interactions among them. If not dealt with carefully, such accidental interactions may lead to behavioral inconsistencies in the execution of the system.

This chapter concludes the dissertation by summarizing our approach to the problem of *achieving a more consistent and predictable behavior of Dynamically Adaptive Software Systems*. Our approach consisted of introducing the formal basis and programming model of CoPNs to support the development of Dynamically Adaptive Software Systems. In this final chapter we review the extent to which our initial research goals were achieved, and repeat the main contributions of this dissertation. We also discuss some limitations of our approach and propose avenues of future work.

11.1 Research Goals Revisited

Section 1.2 stated different situations that can give rise to inconsistencies between the intended and observed behavior of Dynamically Adaptive Software Systems. In view of the identified problems, the principal goal of this dissertation was to provide a comprehensive programming model to *achieve consistency and predictability of dynamically adaptive software systems, in the presence of multiple behavioral adaptations, continuously being introduced to and withdrawn from the system at run time.* To attain this objective we proposed four research goals. In the following we revisit these goals and discuss to what extent they are attained by our proposed formalism and programming model.

G.1: Lack of interaction definition. Chapter 6 introduced context dependency relations as a means to define interactions between adaptations. Context dependency relations are associated with two functions that allow the modification of a CoPN by introducing places, transitions, or arcs. Such modifications are used to describe which context should and should not be (de)activated according to the activation states of other contexts. The set of context dependency relations presented in this dissertation corresponds to the needs identified during the

development of our case studies but does not presume to be complete; more context dependency relations can be added to the current set as needed. The process of defining new context dependency relations was shown in Section 9.3 with the introduction of two new relations.

G.2: Accidental interaction of adaptations. Chapter 6 introduced the process of activating contexts, where interactions between all defined adaptations are dynamically verified through the reactive semantics of CoPNs. Requests to activate or deactivate a context take into account the state of all other contexts defined in the system, by verifying that the constraints defined by the context dependency relations are satisfied. This process first verifies the context dependency relations associated with the context requested for activation, and transitively forwards the request to all other contexts defined in the system. The firing semantics of CoPNs is defined with the objective of avoiding accidental interactions between adaptations. Chapter 7 introduced analyses of the CoPN which identify additional inconsistencies in the definition of context dependency relations. This process uses reachability and liveness analyses to verify that the expected behavior of all context dependency relations defined in the system is as expected. CoPNs cannot yet identify missing context dependency relations. Section 11.4.2 presents a discussion about how the identification of missing context dependency relations could be added to our programming model.

G.3: Lack of verification. Chapter 6 presented the approach introduced in CoPNs for the dynamic verification of context activations. The context dependency constraints that must be satisfied by adaptations are directly encoded in the CoPN representation of the system. Using the firing semantics of reactive Petri nets, dependency constraints are verified at run time. The reactive semantics introduced in CoPNs are used to ensure that activations and deactivations of contexts are always consistent with respect to the defined context dependency relations.

G.4: Lack of property analysis. Chapter 7 presented the property analysis process introduced in CoPNs. CoPNs provide us with the possibility to analyze if the context dependency relations defined in the system are coherent —that is, if there are no interactions between contexts leading to infinite firing steps, or adaptations that can never be activated. The properties that can be reasoned about are reachability of system states and liveness of context activations and deactivations. The analysis process of these properties in CoPNs is supported by an external reasoning engine, LoLA. The CoPN model uses a set of extensions that restrict its analysis capabilities, only allowing the analysis of a restricted semantics of the model. The analysis of system properties is not performed over the complete CoPN but only over a bounded version of it —that is, for each context activations can be verified up to a given (maximum) number.

G.5: Lack of a comprehensive programming model. In the survey of the highly dynamic systems of Chapter 4, we observed how none of the existing CoPN languages provide support for the complete adaptation process proposed in the literature on Dynamically Adaptive Software Systems [159, 41, 27]. In Chapter 8 we presented the CoPN programming model, which complies with the adaptation process for Dynamically Adaptive Software Systems by implementing components for the *discovery*, *analysis* and *management*, *representation*, and *execution* of dynamic adaptations (Figure 8.1). The development of the CoPN programming model focused on the component for managing the consistency between behavioral adaptations. Other components of the process model,

such as the discovery component, are developed as a proof-of-concept, and can still be extended and made more robust. The programming model offered by CoPNs was coupled to the Subjective-C language. Nonetheless, the formal basis of CoPN is independent of a particular programming language, and thus we argue that the programming model could be coupled easily to other COP languages within the same family as Subjective-C (e.g., Ambience [74]), or even more, to other families of COP languages such as layer-based or distributed COP languages (e.g., EventCJ [103] and ContextErlang [71], respectively).

11.2 Contributions

This dissertation presents a formal basis for the specification and development of Dynamically Adaptive Software Systems. In particular we make a contribution to the domain of inconsistency management for such systems. The programming model presented herein is a stepping stone towards a sound definition of more consistent and predictable Dynamically Adaptive Software Systems.

Having explained our approach in detail throughout the dissertation, we now provide a summary describing each of our contributions.

Formalizing COP systems

In this dissertation we developed a formal basis for the definition and execution of Dynamically Adaptive Software Systems, and in particular COP systems. Chapter 6 presented a language and implementation-independent formalization of adaptations and their interactions. Contexts are defined as singleton instances of CoPNs (Definition 6.2). Interactions between adaptations are defined by means of context dependency relations which formally describe how activation and deactivation of contexts interact with other contexts defined (by means of context dependency constraints). COP systems are composed of multiple adaptations represented as a CoPN which consist of all singleton CoPNs with additional places and transitions introduced by the context dependency relations defined in the system (Section 6.2). The purpose of providing COP systems with such a formalization was to define a sound activation semantics of the system. Hence, we defined what it means to be in a consistent state in the system, and how consistent states are preserved as contexts are activated and deactivated dynamically (Section 6.3). Preservation of consistent states is verified at run time as contexts are activated or deactivated using the reactive semantics introduced in CoPNs [33, 30]. Finally, our formalization of COP systems describes how behavioral adaptations are associated with each context, and how they are ordered with respect to the disambiguation techniques implemented in the language. Such formalization describes the selection of the applicable behavioral adaptations for every message send.

Run-time verification of inconsistencies

In Section 4.2 we identified the dynamicity of adaptations as one of the main causes for inconsistencies that may arise in the system. As contexts are activated and deactivated, their associated behavioral adaptations are respectively introduced to and withdrawn from the system. To avoid the problems of *missing* or *interposing* behavioral adaptations at run time, Section 6.2 introduced context dependency relations

as a means to constrain the activation and deactivation of contexts with respect to other defined contexts. Such relations are directly encoded in the execution model of the system —that is, corresponding constraints are represented by transitions and arcs in the CoPN. Context dependency relations are verified at run time as tokens flow through the CoPN, as a result of transition firing. The non fulfillment of context dependency relations is identified in the system by not reaching a consistent state of the CoPN (i.e., reaching a marking in which a temporary place is marked and no internal transition is enabled). As described in Section 6.3.1 inconsistent states are managed by automatically backtracking all changes to the state of the CoPN to its last consistent state according to the reactive semantics of CoPNs [33].

Design-time Identification of inconsistencies

Run-time verification of context dependency relations between contexts is not enough to ensure the correct execution of the system. Even if context activations are guaranteed to be consistent by means of the run-time verification, interactions between contexts could be ill-defined. Chapter 7 introduced a process for the analysis of system properties at design time, allowing us to answer questions as: could a particular set of contexts be active simultaneously? or is it ever possible to activate a context? In particular our approach is targeted at the identification of incoherent definitions of interactions between contexts —that is, states in which the activations of contexts can never happen, or infinite sequences of transition firings can occur. CoPN uses a semiautomated process for the analysis of system properties. For each of the context dependency relations defined in the system, a set of test cases is automatically generated in order to verify the reachability and liveness properties of the system. Such properties are then manually verified using the external analyzer, LoLA. The results of the analyses can be used to reason about structural properties of the CoPN, ensuring the coherence of interactions defined between context.

The semiautomated approach used for the analysis of the system introduced in CoPNs is an improvement over existing COP approaches in the literature (e.g., ContextL [49] or EventCJ [103]). CoPNs do not need to resort to external definitions or formalizations in order to analyze the system's properties, rather CoPNs use the defined context dependency relations to automatically generate the test cases that are used to analyze the reachability and liveness properties of the system.

Comprehensive Programming Model

Chapter 8 presented the general architecture of the CoPN programming model and the details of its implementation. The CoPN programming model was developed with the objective of providing support for the adaptation process and to comply with the general architecture of Dynamically Adaptive Software Systems. As a result we provided a system architecture composed of: (1) A `ContextAcquisition` and a `ContextGathering` module, respectively used for introducing new adaptations and retrieving information about the surrounding execution environment. (2) A `ContextManager` module used for the definition and run-time management of contexts and context dependency relations. (3) A `PetriNetGenerator` module used to unfold CoPNs into Petri nets with place capacities and without inhibitor arcs, and an `AnalysisTestCaseGenerator` module used for generating the analysis test cases. The models generated by these modules are later used as inputs of the LoLA analyzer.

(4) The `ApplicationBehavior` of the system, which is given by the composition of behavioral adaptations associated to the active contexts in the system.

A Formal Basis for the Development of DASS

The overall contribution of this dissertation consists of providing a formal basis for the definition, development, management, and analysis of Dynamically Adaptive Software Systems. The development of the CoPN programming model was based on the case of COP, a highly dynamic class of Dynamically Adaptive Software Systems, as described in Chapter 4. To evaluate the appropriateness of our programming model for the broader class of Dynamically Adaptive Software Systems, we revisited some of the surveyed approaches in Chapter 3 for implementing Dynamically Adaptive Software Systems. Chapter 10 presented a discussion on how CoPNs could be used for the development of other classes of Dynamically Adaptive Software Systems, and how existing challenges could be addressed using CoPNs as a basis for the development of such systems.

11.3 Limitations of CoPNs

The development of the formalism described throughout this dissertation is a contribution to the specification and development of software systems that can consistently adapt their behavior to the changing conditions of the surrounding execution environment. However, we recognize that there are still limitations to the applicability of our work. This section discusses the existing shortcomings of CoPNs described in previous chapters.

11.3.1 A Semantics of the Execution Language

The main goal behind the definition of the CoPN programming model was to provide a more consistent and predictable adaptation of the system's behavior at run time. As such, CoPNs provide a formal specification for defining contexts (Section 6.1), their interactions and composition of contexts defined in a system (Section 6.2.2), the dynamic behavior of the system (Section 6.3.1), the selection of behavioral adaptations (Section 6.4.2) and the scoping of behavioral adaptations (Section 9.2). Nonetheless, we still rely on existing techniques in current COP languages for the execution of behavioral adaptations. Therefore, the CoPN model does not provide a semantics for the execution of behavioral adaptations. This characteristic has been the focus of study in previous approaches targeting the definition of semantics for COP [74, 38, 2, 96, 105].

We argue that, in order to provide a complete formalization of COP systems, the application of behavioral adaptations also needs to be addressed by the system's semantics. An example supporting this statement is the difficulty in defining the resending of behavioral adaptations, which has been proved to cause inconsistencies in layer-based COP languages with dynamic scoping, such as ContextL [39]. Definition of a semantics for the application of behavioral adaptations, and in particular resend of behavioral adaptations, could be used to detect and manage inconsistencies in the application behavior, which is currently not supported in CoPNs.

11.3.2 CoPNs Analysis

In CoPNs interactions between adaptations are modeled by explicitly specifying the allowed and disallowed actions for every context activation and deactivation with respect to the state of other contexts defined in the system. In particular, interactions are modeled using static transition priorities and inhibitor arcs. The use of inhibitor arcs presents a tradeoff in the definition of CoPNs. While using inhibitor arcs allows us to represent conditions in which context can be activated or deactivated due to the inactive state of another context, the use of such arcs turns the different Petri net properties undecidable.

In order to analyze different properties about COP systems other formalisms such as predicate logic could have been used. Nonetheless, such an approach has four major setbacks with respect to the work presented in this dissertation. First, using Boolean logic for the representation of contexts would make it hard to express interaction between adaptations, because it is not possible to model the multiple activation of contexts, required to enable context interaction. Such activation behavior would have to be implemented using a model which is independent to the specification of contexts, for example, using activation counters. Secondly, the expression of all possible combinations of contexts and all interaction rules between contexts need to be supported by external models that verify their completeness. Thirdly, the representation of the system using predicate logic would be useful for the analysis of the system but additional execution model that complies with the logic specification would have to be designed independently. Finally, we argue that having different specifications of the system (i.e., for its analysis, execution, and management of multiple activations) would harness the adoption and development of such systems as more expertise would be required.

Another possibility to deal with the analysis of CoPNs, would be to use over-approximations of the model by means of reset arcs. The idea for such analysis would be to replace inhibitor arcs of the CoPN model with reset arcs [56] and perform the analyses (e.g., coverability) in this new model. In the setting of CoPNs such analysis could be used to ensure that if certain marking is not coverable in the Petri net with reset arcs, it is not coverable in the CoPN. Such results could be used to ensure that it would never be possible to reach illegal states of the system.

These alternatives to ease or extend the analysis of CoPNs would need to be fully developed in order to conduct an in-depth study analyzing their potential advantages and disadvantages with respect to the solution proposed in this dissertation.

11.3.3 Analyses Integration

In Chapter 7 we explored the application of different techniques for the analysis of contexts and context dependency relations, defined at design time. To perform such analyses we used LoLA, a specialized Petri net analyzer. Currently the development and analysis tools are not integrated. This means that developers must manually run each of the test cases generated by CoPNs. Clearly this can become a very time consuming and error-prone task as the system grows, for example, in the case of the Mobile City Guide application described in Section 9.1.2, 1874 test cases were generated which then had to be (manually) verified one by one using LoLA. In addition, the current analysis of system properties can only take place at design time.

It is thus necessary to integrate the development and analysis tools in order to fully automate the analysis process of COP systems, addressing the two aforementioned

issues. Two approaches can be taken for the integration of the tools. One approach would consist of using the libraries available in LoLA for its integration with other tools.[1] Such integration, however, restricts the tools to be compatible with C, the language in which LoLA is developed. Although it is still possible to use the library with other languages, for example, with Java as it is the case of ePNK [112], this imposes a technological infrastructure which may not be acceptable for the deployment of CoPNs in sensor networks or mobile devices. The second approach would consist of creating a tailored analysis of the system's properties for CoPNs, for example using a coverability-tree like reduction techniques [80, 95]. This approach answers to a current limitation of using LoLA as an analysis tool. Since lola specializes in the analysis of place/transition nets, we must analyze CoPNs without their reactive or static priorities semantics. Maintaining the semantics of CoPNs would improve the process of analyzing their properties.

11.3.4 Introduction of Behavioral Adaptations and Context Dependency Relations

Section 8.1.4 presented the proof-of-concept implementation of the context acquisition module in CoPNs. Our programming model currently maintains a single centralized CoPN for the management of the system's contexts. The context acquisition module allows the introduction of new (unknown) contexts to this CoPN, and the definition of context dependency relations between its contexts. Introduction of behavioral adaptations, however, is not yet supported in CoPNs, because the underlying mechanism for the introduction of behavioral adaptations used by Subjective-C does not allows it. In Subjective-C behavioral adaptations are only processed and included as part of an application at compile time. This is a current limitation detected in all existing COP languages.

Nonetheless, we recognize that in order to provide a true context acquisition module it is necessary to enable the introduction of complete adaptations (i.e., contexts and their associated behavioral adaptations), instead of only allowing the introduction of contexts. Introduction of behavioral adaptations would also require a run-time analysis and verification of the introduced behavior, which is not currently possible in the CoPN model.

Definition of behavioral adaptations in CoPNs currently follows the approach introduced in Subjective-C [123]. In Subjective-C each behavioral adaptation is processed at compile time, creating the corresponding `SCMethod` as defined in Snippet 4.9. Subjective-C specific methods keep a reference to the base implementation of the method and the behavioral adaptation. A similar definition of behavioral adaptations could be used for the dynamic acquisition of behavioral adaptations, for example, by means of block constructs.[2] Blocks, similar to *closures* [1], are first-class anonymous functions which can capture and modify the state of the lexical scope within which they are defined. Such capabilities of blocks would allow the introduction behavioral adaptations alongside their associated contexts. Blocks could also aid in the definition and introduction of new types of context dependency relations, where the definition

[1] Cf. LoLA presentation available at: `http://www.slideshare.net/correctsystems/verification-with-lola-6-integrating-lola`

[2] `http://developer.apple.com/library/ios/#documentation/cocoa/Conceptual/Blocks/Articles/00_Introduction.html`

of dependency constraints would be described within the block's behavior, allowing the introduction of new definitions of $\texttt{ext}_R$ and $\texttt{cons}_R$ functions.

11.4 Future Work

In this section we discuss how the work presented in this dissertation could be extended and complemented, providing avenues for future research.

11.4.1 Studying Alternatives to CoPNs

In this dissertation we opted for a Petri net-based model for the modeling, analysis and execution of Dynamically Adaptive Software Systems. However, we recognize that other formalizations (e.g., statecharts, Boolean logic) could be used to model, analyze and execute such systems. An interesting avenue of future work would be to develop such formalisms in order to generate a qualitative comparison of the different approaches.

The development of execution models for Dynamically Adaptive Software Systems using other formalisms requires additional analysis about the reasoning capabilities offered by the formalism. The defined interactions between contexts yield a a Turing complete model rendering desired properties of the model undecidable. We would required to analyze if the verification mechanisms provided by other formalisms enabled an analysis approximation to deal with such types of systems.

11.4.2 Behavior Analysis of Dynamically Adaptive Software Systems

Dynamically Adaptive Software Systems modify the control flow of the system during its execution with the introduction of behavioral adaptations. This makes software systems programmed in such a style difficult to follow and reason about. In this dissertation we addressed the problem of inconsistent behavior of dynamic adaptations from the view point of the interactions between the contexts associated to behavioral adaptations. Introduced behavioral adaptations could be inconsistent between each other, even if they respect the interactions between their associated contexts. We see an opportunity for future research in the consistency management of behavioral adaptations. Such an approach would require, similarly to the work presented in this dissertation, a means to manage and analyze behavioral adaptations. Three main research tracks could steer future research in the analysis of Dynamically Adaptive Software Systems.

Completeness of Behavioral Adaptations

A first avenue of research for the analysis of behavioral adaptations consists of a design-time analysis to reason about the completeness of behavioral adaptations. The notion of **completeness** has been defined in the perspective of predicate dispatch to "guarantee that no *'message not understood'* error is raised —for every possible set of arguments at each call site, some method is applicable" [60]. COP languages do not provide any guarantees about completeness of behavior. We believe this notion

of completeness could be used to reason about behavioral adaptations defined in the system.

Two aspects need to be considered to analyze the completeness of the system's behavior: the definition of behavioral adaptations, and the activation state of their associated contexts; fortunately, such information is already provided by the CoPN model. Each context defined in the system is aware of all of its associated behavioral adaptations. The activation state of contexts is available from the context representation module —that is, the current marking of the system. Additionally, activation states of contexts could be deduced from the context dependency relations defined between adaptations. For example, taking the case of an implication dependency relation, we know that every time, the source context of the relation is active, the target context is also active. In such a case, the behavioral adaptations provided by the target context, are available to use by the source context. Such information would allow us to construct a **behavior interaction graph** for the system —that is, a graph representing the concrete behavioral adaptations (implementations), and the different situations in the surrounding execution environment of the system in which such behavior can be used.

Such an approach for reasoning about the system behavior could be used to further increase the predictability of the system and avoid behavior incompleteness, and hence inconsistencies.

Identifying Conflicts Between Behavioral Adaptations

A second approach to the analysis of behavioral adaptations consists of a compile-time analysis of the behavioral adaptations definition. The idea behind this approach is to build a **context control flow graph**, a control flow graph [157] that branches out in all behavioral adaptations of every method.

The purpose of this approach is to identify different types of behavioral inconsistencies. For example, navigating through all branches of behavioral adaptations for a particular method could be used to identify behavioral adaptations that provide contradicting behavior, or are missing behavior to finish their execution correctly.

Construction of a control flow graph for Dynamically Adaptive Software Systems does not come without challenge, as it has not yet been attempted. At compile time it is unknown which behavioral adaptations of a method will be executed, hence the graph will contain many nodes with multiple outgoing edges, for example, at every method with multiple behavioral adaptations. As a result, there will be many possible paths through this graph, which may not scale, making it difficult to identify paths in which conflicting behavior might be active. State explosion of the context control flow graph could be reduced, for example, using the interaction information provided by context dependency relations

By analyzing the source code of the application, it is possible to determine which behavioral adaptations will never be adopted at the same time. Hence, CoPNs do not require to check for conflicts between these behaviors at run time, reducing the performance overhead of such verification.

Test Case Refinement Using Behavioral Inconsistencies

Using information gathered from the behavior analysis of the system (by means of either of the aforementioned approaches) it would be possible to expand the test cases currently generated by CoPNs. Chapter 7 discussed the process of generating a series

of test cases with CoPNs, and their later analysis using LoLA. These test cases are based on the structure provided by the context dependency relations defined in the system. Using information about the behavioral adaptations defined in the system could be helpful to refine and expand the generated test cases.

11.4.3 Extending Dynamically Adaptive Software Systems

With respect to the development of and reasoning about Dynamically Adaptive Software Systems there are still multiple open questions to be answered. Here we discuss two challenges that became apparent in the development of our approach.

Concurrent Execution of Dynamically Adaptive Software Systems

Adaptations to a system's behavior are normally processed sequentially. The Petri net model, however, has been used in the modeling of concurrent process since its early origins [156], it is thus natural to ask ourselves if the programming model presented in this dissertation could be opened to a concurrent computational model.

Opening the programming model to a concurrent evaluation of the system's instructions and context activations would require revisiting the execution model of behavioral adaptations, but more importantly it would require us to revisit the assessment and verification of consistency. Since requests to activate or deactivate adaptations can take place concurrently, the order in which transitions are fired may interfere with the firing of other enabled transitions —that is, firing transitions enabled by one activation request may disable the firing of transitions enabled by other activation requests. In order to adopt a concurrent evaluation of context activations the semantics for their evaluation would need to be redefined. Similarly, based on the new execution model, the activation semantics of the CoPN would need to be revisited in the light of Reduction Rules 6.19 through 6.23 of the adaptation activations semantics, and Theorem 6.3. Process algebra formalisms such as the box algebra [17] could be further studied in order to pursue a concurrent execution of COP systems.

Temporal execution of Dynamically Adaptive Software Systems

As a means to push further the dynamicity and adaptability of Dynamically Adaptive Software Systems it would be interesting to allow the programming model to reason about temporal properties of the system. So far adaptations are defined by means of **contexts** —situations in the surrounding execution environment that are semantically meaningful for the system. However, during the exploration of Dynamically Adaptive Software Systems and the development of different case studies, we identified cases in which adaptations could take place due to past (or future) events, rather than taking place to a particular situation currently present in the surrounding execution environment. For example, we could use these temporal conditions for the introduction of behavioral adaptations that use a particular variable, only when the variable has been given a concrete value.

In order to extend the context activation mechanism to take into account temporal conditions of the system we could take inspiration from timed Petri nets [140] and modal logic [19]. Timed Petri nets allow us to annotate transitions with an interval of time for which the transition is enabled once all its input places are marked. Using a similar approach it would be possible to annotate transitions, not with a time interval,

but rather with a temporal formula describing conditions about the execution of the system.

11.4.4 Extensions and Uses of CoPNs

The work presented in this dissertation is a contribution for the development of Dynamically Adaptive Software Systems. However we recognize different points of extension or improvement which the programming model could profit from, to facilitate the development of Dynamically Adaptive Software Systems.

Automated Analysis of LoLA Outputs

The semi-automated process for the analysis of system properties presented in Chapter 7 consists of in the automated generation of different test cases, and their later verification using LoLA. However, some of the generated test cases (e.g., case 1. of the causality dependency relation defined in Section 7.2.1) requires further analysis of the outputs generated by LoLA —that is, analysis of the generated firing sequence or reached state.

In order to analyze the output state of the analysis it is required to generate the expected state of the CoPN and compare it with the output state provided by the analysis. The analysis would be accepted if the only marked places are those generated with the test case, and the marking of each reached place is greater than or equal to the marking provided in the test case. Note that the process to compare the generated output states needs to be implemented, as it is not supported by LoLA.

The analysis of output firing sequences requires more work. To analyze if the output firing sequence is a valid step of the CoPN we could re-use the ideas of the reachability tree analysis [107, 80, 95] to reason about the sequences of transition firings, under the firing semantics of CoPNs. Our firing sequence tree analysis consists of building a tree of valid sequences of transition firings for a given marking, where each node of the tree corresponds to the set of enabled transitions, and each edge corresponds to the firing of a particular transition at each node. Once the tree is constructed, we are interested in analyzing all possible permutations of consistent steps from the output firing sequence given by LoLA. For each of the consistent steps a tree of transitions firings is created, analyzing if that particular step is possible in the CoPN —that is, if firing the transitions of the step leads to a leaf of the tree, where no (internal) transition is enabled.

Method Dispatching with CoPNs

CoPNs were presented in Chapter 8 as a programming model that successfully covers the complete adaptation process of Dynamically Adaptive Software Systems through the context-awareness architecture. In our discussion of the CoPN programming model we delegated the selection of behavioral adaptations to the method dispatching mechanism implemented by the underlying COP language (Subjective-C). However, we noticed that the CoPN model already contains all the required information to select the applicable behavioral adaptations of the system.

The process of the method dispatching mechanism of Dynamically Adaptive Software Systems supported by CoPNs could be defined as follows:

1. Message sends that have behavioral adaptations are looked up for active contexts associated with the behavioral adaptations. If the message has no behavioral adaptations, then it is resolved by the default context.
2. Active contexts are ordered according to the disambiguation techniques available in the system, following Definition 6.27. The responder to the message would be the first behavioral adaptation in the order.
3. Reuse of behavior through the `@resend()` language construct is treated as regular message send. Whenever a `@resend` is sent, behavioral adaptations are ordered as described in step 2. However, such process differs from a simple message send in the sense that the responder to the message would be the first behavioral adaptation which is an immediate successor, with respect to the given order, to the behavioral adaptation with which the `@resend` message originated.

CoPNs Tool Support

Currently there is little tool support for the development of COP applications, other than that provided by the underlying programming language of the system. In many cases integration between the COP language and the tools provided by the underlying programming model is not evident or even possible. Adaptation of the system's behavior as proposed by COP constitutes a paradigm shift from common Object-Oriented Programming, and thus it also requires specialized tools to support its development.

In this dissertation we provided a step forward in this direction, by defining a tool that allows the manipulation and visualization of contexts and their states. Additionally, we provided a tool for the simulation of context activations that allows us to observe interaction between adaptations. The reach of such a tool could be expanded from a support tool for the simulation of the system, to a tool for its development.

The visualization of the CoPN provided in Figure 8.7 could be reused for the development and definition of adaptations by modifying its behavior from a SIMULATION mode to a DEVELOPMENT mode as follows:

- The firing steps view could be used to inspect and add new behavioral adaptations to a given context. Behavioral adaptations associated with the context would be presented as a list of methods.
- The context information view could be used for the visualization of the implementation of behavioral adaptations. Every time a behavioral adaptation is selected in the firing steps view, its corresponding implementation would be displayed in the context information view.
- The CoPN view would be used to display contexts as is currently done. However the behavior for highlighting contexts would change. Only one context could be selected at a time, providing a list of its behavioral adaptations in the firing steps view and the details of the implementation for a particular behavioral adaptation in the context information view as described earlier. Other contexts in the CoPN view could be highlighted automatically (for example in red or green) as a means to show the contexts which also define behavioral adaptations for (a subset of) the behavior that the selected context does. Highlighting of contexts could be based on the outputs of the behavioral adaptations analysis as envisioned in Section 11.4.2, for example, conflicting contexts would be highlighted in red.

Using the CoPN-IDE tool as a means for the development of Dynamically Adaptive Software Systems is a challenging task. This extension requires the development of an interpreter or compiler to be able to generate executable code and deploy applications. Nonetheless, we see the value in developing such a tool, because it would unify the development of different Dynamically Adaptive Software Systems under one single environment. Clearly, other tools supporting the development of the system, such as specialized debuggers for dynamic adaptations, could still be integrated with the CoPN-IDE tool.

Performance Improvements

In Section 9.4 we observed that the run-time verification of consistency in COP systems has an important impact on the overall performance of the system's execution. A pre-analysis and cache of context activations could be performed to reduce the performance overhead of the run-time verification.

The CoPN structure presented in this dissertation is static, apart from the marking. Using the static structure of the CoPN —that is, the initial set of adaptations and context dependency relations defined between them, it is possible to pre-calculate the context activations, in order to avoid the run-time verification of every activation or deactivation. Using the existing reachability analysis described in Chapter 7, it would be possible to re-use the generated *reachability tree* for the CoPN to avoid verification of every activation or deactivation. At run time, context activations are not resolved by means of the activation semantics of the CoPN, but rather they would be resolved directly by a cache structure, or production system, consisting of all possible activations or deactivations that lead to a consistent state under a specific configuration of active contexts.

This performance optimization technique would need to be revisited in the case the structure of the CoPN is modified at run time. If new adaptations are introduced, or context dependency relations between adaptations are defined the constraints of activation and deactivation of an adaptation may change. It is required to study whether the reachability tree can be (incrementally) generated at run time, or whether the cache structure can be filled as contexts are activated and deactivated.

Bibliography

The references are sorted alphabetically by first author.

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1996. ISBN 0-262-01153-0. Second Edition.

[2] Tomoyuki Aotani, Tetsuo Kamina, and Hidehiko Masuhara. Featherwight eventcj: A core calculus for a context-oriented language with event-based per-instance layer transition. In *Proceedings of the 3rd Context-Oriented Programming Workshop*, COP'11, pages 1–7. ACM, July 2011.

[3] M. Appeltauer, R. Hirschfeld, and T. Rho. Dedicated programming support for context-aware ubiquitous applications. In *International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*, UBICOMM'08, pages 38–43. IEEE Computer Society, October 2008.

[4] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. A comparison of context-oriented programming languages. In *International Workshop on Context-Oriented Programming*, pages 1–6, New York, NY, USA, 2009. ACM Press. ISBN 978-1-60558-538-3. DOI 10.1145/1562112.1562118.

[5] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, and Hidehiko Masuhara. ContextJ: Context-oriented programming for Java. *Computer Software of The Japan Society for Software Science and Technology*, 6 2010.

[6] Malte Appeltauer, Robert Hirschfeld, Hidehiko Masuhara, Michael Haupt, and Kazunori Kawauchi. Event specific software composition in context oriented programming. In *proceedings of the Conference on Software Composition*, pages 50 – 65. Springer, July 2010.

[7] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. Transactions on aspect-oriented software development i. chapter An overview of caesarj, pages 135–173. Springer-Verlag, Berlin, Heidelberg, 2006.

[8] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, April 2010.

[9] Engineer Bainomugisha. *Reactive Method Dispatch for Ocntext-Oriented Programs*. PhD thesis, Vrije Universiteit Brussel, Brussels, Belgium, December 2012.

[10] Engineer Bainomugisha, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. A survey on reactive programming. *ACM Computing Surveys*, 2012.

[11] Engineer Bainomugisha, Jorge Vallejos, Coen De Roover, Andoni Lombide Carreton, and Wolfgang De Meuter. Interruptible context-dependent executions: A fresh look at programming context-aware applications. In *In Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software Proceedings*, OnWard'12, Tucson, Arizona - USA, October 2012. ACM.

[12] Falko Bause. On the analysis of petri nets with static priorities. In *Acta Informatica*, volume 33, pages 669 – 685, 1996.

[13] Falko Bause. Analysis of petri nets with a dynamic priority method. In *Proceedings of the International Conference on Application and Theory of Petri Nets*, ICATPN'97, pages 215–234, Toulouse, 1997. Springer-Verlag.

[14] Gerard Berry, Georges Gonthier, Ard Berry Georges Gonthier, and Place Sophie Laltte. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[15] Eike Best. Fairness and conspiracies. *Information Processing Letters*, 18(4): 215–220, May 1984.

[16] Eike Best and Maciej Koutny. Petri net semantics of priority systems. *Theoretical Computer Science*, 96:175–215, April 1992. ISSN 0304-3975. URL `http://dx.doi.org/10.1016/0304-3975(92)90184-H`.

[17] Eike Best, Raymond Devillers, and Maciej Koutny. The box algebra=petri nets+process expressions. *Information and Computation*, 178(1):44 – 100, 2002.

[18] Olivier Biberstein, Didier Buchs, and Nicolas Guelfi. Object-oriented nets with algebraic specifications: The co-opn/2 formalism. In *Advances in Petri Nets on Object-Orientation: Lecture Notes in Computer Science*, pages 73–130. Springer-Verlag, 1997.

[19] Patrick Blalckburn, Maarten de Rijke, and Yde Venema. *Modal Logic*. Cambridge University Press, 2001.

[20] Andrea Bobbio. System modeling with petri nets. *System Reliability Assessment*, pages 102 – 143, 1990.

[21] Daniel Bobrow, Linda DeMichiel, Richard Gabriel, Sonya Keene, Gregor Kiczales, and David Moon. Common Lisp Object System specification. *Lisp and Symbolic Computation*, 1(3/4):245–394, 1989.

[22] Eric Bodden, Patrick Lam, and Laurie Hendren. Clara: a framework for partially evaluating finite-state runtime monitors ahead of time. In *International Conference on Runtime Verification*, volume 6418, pages 74 – 88. LNCS, 2010.

[23] Egon Börger and Robert Stärk. *Abstract State Machines*. Springer-Verlag, 2003.

[24] Andreas Brodt, Daniela Nicklas, Sailesh Sathish, and Bernhard Mitschang. Context-aware mashups for mobile devices. In *Proceedings of the 9th international conference on Web Information Systems Engineering*, WISE '08, pages 280–291, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-85480-7.

[25] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, and Michael Stal. *Pattern-Oriented Software Architecture: A system of Patterns*, volume 1. Wiley, August 1996. ISBN 978-0471958697.

[26] Nadia Busi. Analysis of petri nets with inhibitor arcs. *Theoretical Computer Science*, 275:127 – 177, February 2002.

[27] Alfredo Cádiz, Sebastián González, and Kim Mens. Orchestrating context-aware systems: A design perspective. In *Proceedings of the First International Workshop on Context-Aware Software Technology and Applications*, pages 5–8. ACM Press, 2009. ISBN 978-1-60558-707-3. DOI 10.1145/1595768.1595771. Co-located with ESEC/FSE.

[28] Muffy Calder, Mario Kolberg, Magill Evan H., and Stephan Reiff-Marganiec. Feature interaction: A critical review and considered forecast. *Computer Networks*, 41:115–141, 2003.

[29] T-H Cao and A. C. Sanderson. *Intelligent Task Planning Using Fussy Petri Nets*, volume 3 of *Series in Intelligent Control and Intelligent Automation*. World Scientific, August 1996.

[30] Nicolás Cardozo, Sebastián González, Kim Mens, Ragnhild Van Der Straeten, and Theo D'Hondt. Modeling and analyzing self-adaptive systems with context petri nets. In *Proceedings of the Symposyum on Theoretical Aspects of Software Engineering*, TASE'13, pages 191 – 198, Birmingham, UK, July 2013. IEEE Computer Society.

[31] Nicolás Cardozo, Sebastián González, Kim Mens, and Theo D'Hondt. Safer context (de)activation through the prompt-loyal strategy. In *Proceedings of the International Workshop on Context-Oriented Programming*, COP'11, pages 2:1–2:6. ACM Press, 2011. ISBN 978-1-4503-0891-5. DOI 10.1145/2068736.2068738. 25 July 2011. Co-located with ECOOP.

[32] Nicolás Cardozo, Sebastián González, and Kim Mens. Uniting global and local context behavior with context petri nets. In *Proceedings of the International Workshop on Context-Oriented Programming*, number 3 in COP'12, pages 1–3. ACM Press, 2012. DOI 10.1145/2307436.2307439. 11 June 2012. Co-located with ECOOP.

[33] Nicolás Cardozo, Jorge Vallejos, Sebastián González, Kim Mens, and Theo D'Hondt. Context Petri nets: Enabling consistent composition of context-dependent behavior. In Lawrence Cabac, Michael Duvigneau, and Daniel Moldt, editors, *Proceedings of the International Workshop on Petri Nets and Software Engineering*, volume 851 of *CEUR Workshop Proceedings*, pages 156–170. CEUR-WS.org, 2012. 25–26 June 2012. Co-located with the International Conference on Application and Theory of Petri Nets and Concurrency.

[34] Carlos Cetina, Pau Giner, Joan Fons, and Vicente Pelechano. Autonomic computing through reuse of variability models at runtime: The case of smart homes. *Computer*, 42(10):37–43, 2009.

[35] Carlos Cetina, Øystein Haugen, Xiaorui Zhang, Franck Fleurey, and Vicente Pelechano. Strategies for variability transformation at run time. In *Proceedings of the International Software Product Line Conference*, pages 61–70, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[36] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Don Carney, Uğur Çetintemel, Ying Xing, and Stan Zdonik. Scalable distributed stream processing. In *In Proceedings of the Conference on Innovative Data Systems Research*, CIDR'03, 2003.

[37] Giovanni Chiola, Susanna Donatelli, and Guiliana Franceschinis. Priorities, inhibitor arcs, and concurrency in P/T nets. In *Proceedings of the 12th International Conference on Application and Theory of Petri Nets*, pages 182–205, jun 1991.

[38] Dave Clarke and Ilya Sergey. A semantics for context-oriented programming with layers. In *International Workshop on Context-Oriented Programming*, number 10 in COP'09, pages 1 – 6. ACM, July 2009.

[39] Dave Clarke, Pascal Costanza, and Éric Tanter. How should context-escaping closures proceed? In *International Workshop on Context-Oriented Programming*, COP '09, pages 1–6. ACM, 2009. ISBN 978-1-60558-538-3. DOI 10.1145/1562112.1562113.

[40] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263, April 1986.

[41] Andreas Classen, Arnaud Hubeaux, Frans Sanen, Eddy Truyen, Jorge Vallejos, Pascal Costanza, Wolfgang De Meuter, Patrick Heymans, and Wouter Joosen. Modeling variability in self-adaptive systems: Towards a research agenda. In *Composition and Generative Techniques for Product Line Engineering (McGPLE'08)*, October 2008.

[42] Gregory H. Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *Proceedings of the 15th European conference on Programming Languages and Systems*, ESOP'06, pages 294–308, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-33095-X, 978-3-540-33095-0.

[43] Pascal Costanza and Theo D'Hondt. Feature descriptions for context-oriented programming. In Steffen Thiel and Klaus Pohl, editors, *International Software Product Line Conference, Second Volume (Workshops)*, pages 9–14. Lero Int. Science Centre, University of Limerick, Ireland, 2008. ISBN 978-1-905952-06-9. 2nd International Workshop on Dynamic Software Product Lines (DSPL'08).

[44] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming: An overview of ContextL. In *Proceedings of the Dynamic Languages Symposium*, pages 1–10. ACM Press, October 2005. DOI 10.1145/1146841.1146842. Co-located with OOPSLA'05.

[45] Pascal Costanza and Robert Hirschfeld. Reflective layer activation in ContextL. In *Proceedings of the ACM symposium on Applied computing*, pages 1280–1285. ACM Press, 2007. ISBN 1-59593-480-4. DOI 10.1145/1244002.1244279.

[46] Sylvain Degrandsart, Serge Demeyer, Jan Van den Bergh, and Tom Mens. A transformation-based approach to context-aware modelling. *Software & Systems Modeling*, pages 1–18, 2012.

[47] Jörg Desel and Wolfgang Reisig. *Lectures on Petri Nets I: Basic Models*, volume 1491 of *LNCS*, chapter Place/transition Petri Nets, pages 122–173. Springer, 1998.

[48] Brecht Desmet, Jorge Vallejos, and Pascal Costanza. Introducing mixin layers to support the development of context-aware systems. In *3rd European Workshop on Aspects in Sofware (EWAS 2006)*, August 2006.

[49] Brecht Desmet, Jorge Vallejos, Pascal Costanza, Wolfgang De Meuter, and Theo D'Hondt. Context-oriented domain analysis. In Boicho Kokinov, Daniel C. Richardson, Thomas R. Roth-Berghofer, and Laure Vieu, editors, *Modeling and Using Context*, Lecture Notes in Computer Science, pages 178–191, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-74254-8. DOI 10.1007/978-3-540-74255-5_14.

[50] Brecht Desmet, Jorge Vallejos, Pascal Costanza, and Robert Hirschfeld. Layered design approach for context-aware systems. In *Proceedings of 1st International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2007)*, pages 157–165. Technical Report at Irish Software Engineering Research Centre (Lero), January 2007.

[51] Brecht Desmet, Kristof Vanhaesebrouck, Jorge Vallejos, Pascal Costanza, and Wolfgang De Meuter. The puzzle approach for designing context-enabled applications. In *XXVI International Conference of the Chilean Computer Science Society*, pages 23 – 29. IEEE, 2007.

[52] Raymond Devillers, H. Klaudel, and M. Koutny. Context-based process algebras for mobility. In *Proceedings of the 4th International Conference on Application of Concurrency to System Design*, ACSD'04. IEEE Computer Society, 2004.

[53] Anind K. Dey. Understanding and using context. *Personal Ubiquitous Computing*, 5(1):4–7, January 2001.

[54] Michel Diaz. *Petri Nets Fundamental Models Verification and Applications*. Control Systems, robotics and Manufacturing. Wiley, Hoboken, NJ, USA, 3 edition, 2009.

[55] Cristian Dimitrovici, Udo Hummert, and Laure Petrucci. Semantics, composition and net properties of algebraic high-level nets. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1991*, volume 524 of *Lecture Notes in Computer Science*, pages 93–117. Springer Berlin / Heidelberg, 1991.

[56] C. Dufourd, A. Finkel, and Ph. Schnoebelen. Reset nets between decidability and undecidability. In KimG. Larsen, Sven Skyum, and Glynn Winskel, editors, *Automata, Languages and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 103–115. Springer Berlin Heidelberg, 1998.

[57] Lyn Dupré. *BUGS in Writing: A Guide to Debugging Your Prose*. Addison-Wesley Longman Publishing Co., Inc., revised edition, 1998. ISBN 0-201-37921-X.

[58] M. H. Van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23:569–574, 1976.

[59] Thomas Erl. *Service-oriented architecture*. Prentice Hall Englewood Cliffs, 2004.

[60] Michael D. Ernst, Craig S. Kaplan, and Craig Chambers. Predicate dispatching: A unified theory of dispatch. In *ECOOP '98, the 12th European Conference on Object-Oriented Programming*, pages 186 – 211, July 1998.

[61] Rik Eshuis and Juliane Dehnert. Reactive petri nets for workflow modeling. In *Application and Theory of Petri Nets 2003*, pages 296–315. Springer, 2003.

[62] Javier Esparza and Mogens Nielsen. Decidability issues for petri nets. Technical Report RS-94-8, BRICS, Aarhus, Denmark, May 1994.

[63] Dirk Fahland, Cedric Favre, Barbara Jobstmann, Jana Koehler, Niels Lohmann, Hagen Völzer, and Karsten Wolf. Instantaneous soundness checking of industrial business process models. In Umeshwar Dayal, Johann Eder, Jana Koehler, and Hajo A. Reijers, editors, *Business Process Management, 7th International Conference*, volume 5701 of *LNCS (BPM'09)*, pages 278–293. Springer-Verlag, September 2009.

[64] Antonio Filieri, Carlo Ghezzi, and Giordano Tamburrelli. Run-time efficient probabilistic model checking. In *Proceedigns of the International Conference on Software Engineering*, ICSE'11, pages 341–350, Waikiki, HI, USA, May 2011.

[65] Charles Lanny Forgy. *On the efficient implementation of production systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1979.

[66] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 978-0321127426.

[67] Martin Fowler. Inversion of control containers and the dependency injection pattern, January 2004. URL `http://www.martinfowler.com/articles/injection.html`.

[68] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0-201-63361-2.

[69] Michael L. Gassanenko. Context-oriented programming: Evolution of vocabularies. In *Proceedings of the euroForth Conference*, 1993.

[70] Michael L. Gassanenko. Context-oriented programming. In *Proceedings of the euroForth Conference*, April 1998.

[71] Carlo Ghezzi, Matteo Pradella, and Guido Salvaneschi. Programming language support to context-aware adaptation: a case-study with Erlang. In *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS'10, pages 59–68, New York, NY, USA, 2010. ACM Press. ISBN 978-1-60558-971-8. DOI 10.1145/1808984.1808991.

[72] Carlo Ghezzi, Matteo Pradella, and Guido Salvaneschi. An evaluation of the adaptation capabilities in programming languages. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pages 50–59, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0575-4.

[73] Sebastián González, Kim Mens, Marius Colacioiu, and Walter Cazzola. Context traits: Dynamic behaviour adaptation through run-time trait recomposition. In *Proceedings of the International conference on Aspect Oriented Software Development*, MODULARITY.AOSD'13, Fukuoka, Japan, March 2013.

[74] Sebastián González. *Programming in Ambience: Gearing Up for Dynamic Adaptation to Context*. PhD thesis, Université catholique de Louvain, October 2008. URL `http://hdl.handle.net/2078.1/19684`. Coll. EPL 211/2008. Promoted by Prof. Kim Mens.

[75] Sebastián González, Kim Mens, and Patrick Heymans. Highly dynamic behaviour adaptability through prototypes with subjective multimethods. In *Proceedings of the Dynamic Languages Symposium*, pages 77–88. ACM Press, October 2007. ISBN 978-1-59593-868-8. DOI 10.1145/1297081.1297094. Co-located with OOPSLA'07.

[76] Sebastián González, Kim Mens, and Alfredo Cádiz. Context-Oriented Programming with the Ambient Object System. *Journal of Universal Computer Science*, 14(20):3307–3332, 2008. ISSN 0948-6968. DOI 10.3217/jucs-014-20-3307.

[77] Sebastián González, Nicolás Cardozo, Kim Mens, Alfredo Cádiz, Jean-Christophe Libbrecht, and Julien Goffaux. Subjective-C: Bringing context to mobile platform programming. In Brian Malloy, Steffen Staab, and Mark van den Brand, editors, *Proceedings of the International Conference on Software Language Engineering*, volume 6563 of *Lecture Notes in Computer Science*, pages 246–265. Springer-Verlag, 2011. ISBN 978-3-642-19439-9. DOI 10.1007/978-3-642-19440-5_15.

[78] Deepak Gupta. *On-Line Software Version Change*. PhD thesis, Indian Institute of Technology, Kanpur, India, November 1994.

[79] P. J. Haas and G. S. Shedler. Stochastic petri nets: Modelling power and limit theorems. *Probability in the Engineering and Informational Sciences*, pages 477–498, 1991.

[80] M. Hack. Decision problems for petri nets and vector addition systems. Technical Report 95-1, Massachussets Institute of Technology, Cambridge, MA, USA, August 1974.

[81] Philipp Haller and Martin Odersky. Event-based programming without inversion of control. In *Modular Programming Languages*, pages 4–22. Springer, 2006.

[82] S. Hallsteinsen, M. Hinchey, Sooyong Park, and K. Schmid. Dynamic software product lines. *Computer*, 41(4):93–95, April 2008.

[83] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231 – 274, 1987.

[84] N Harvey and J Morris. Nl: A parallel programming visual language. *Australian Computing*, 28(1):2–12, 1996.

[85] Christopher T Haynes, Daniel P Friedman, and Mitchell Wand. Obtaining coroutines with continuations. *Comput. Lang.*, 11(3-4):143–153, January 1986.

[86] Matthew Hennessy. *Algebraic Theory of Processes*. The MIT Press, Cambridge, Mass., 1988.

[87] Michael Hicks and Scott M. Nettles. Dynamic software updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(6):1049 – 1096, November 2005.

[88] Sebastian Hinz, Karsten Schmidt, and Christian Stahl. Transforming bpel to petri nets. In Wil M. P. van der Aalst, B. Benatallah, F. Casati, and F. Curbera, editors, *Proceedings of the Third International Conference on Business Process Management*, volume 3649 of *LNCS (BPM'05)*, pages 220–235, September 2005.

[89] Robert Hirschfeld, Pascal Costanza, and Michael Haupt. An introduction to context-oriented programming with ContextS. In *Generative and Transformational Techniques in Software Engineering II*, volume 5235 of *Lecture Notes in Computer Science*, pages 396–407, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-88642-6. DOI 10.1007/978-3-540-88643-3_9.

[90] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, March–April 2008. URL `http://www.jot.fm/issues/issue_2008_03/article4/`.

[91] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 2 edition, June 2004.

[92] A. Holzer, L. Ziarek, K.R. Jayaram, and Patrick Eugster. Abstracting context in event-based software. *Transactions on Aspect-Oriented Software Development*, page to appear, 2012.

[93] Gerard J. Holzmann. The model checker spin. *IEEE Transactions of Software Engineering*, 23(5):279–295, May 1997.

[94] John E. Hopcroft and Jefferey Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[95] Peter Huber, Arne M. Jensen, Leif O. Jepsen, and Kurt Jensen. Reachability trees for high-level petri nets. *Theor. Comput. Sci.*, 45(3):261–292, 1986.

[96] Atsushi Igarashi, Robert Hirschfeld, and Hidehiko Masuhara. A type system for dynamic layer composition. In *Proceedings of 19th International Workshop on Foundations of Object-Oriented Languages*, FOOL'12, pages 13–24. FOOL online proceedings, October 2012.

[97] Bart Jacobs. Exercises in coalgebraic specification. In *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, pages 237–280, 2000.

[98] Julian Janssens. Scopj: A java framework for context-oriented programming on android. Master's thesis, Université catholique de Louvain, June 2011.

[99] Kurt. Jensen and Lars M. Kristensen. *Coloured Petri Nets: Modeling and validation of Concurrent Systems*. Springer-Verlag, 2009.

[100] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1):1–34, March 2004.

[101] Jevgeni Kabanov. Jrebel tool demo. *Electronic Notes in Theoretical Computer Science*, 264(4):51 –57, February 2011.

[102] Guillaume Kaisin. Engineering context-oriented applications. Master's thesis, Université catholique de Louvain, Louvain-la-Neuve, Belgium, June 2012.

[103] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. Eventcj: A context-oriented programming language with declarative event-based context transition. In *Proceedings of the International Conference on Aspect-Oriented Software Development*, AOSD'11, pages 253–264. ACM Press, March 2011.

[104] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. Bridging real-world contexts and units of behavioral variations by composite layers. In *Proceedings of the International Workshop on Context-Oriented Programming*, COP '12, pages 4:1–4:6, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1276-9.

[105] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. A core calculus of composite layers. In *Proceedings of the 12th workshop on Foundations of aspect-oriented languages*, FOAL '13, pages 7–12, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1865-5.

[106] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-21, Carnegie-Mellon University Software Engineering Institute, November 1990.

[107] Richard M. Karp and Raymond E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3(2):147 – 195, 1969.

[108] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE'08, pages 311–320, New York, 2008. ACM. ISBN 978-1-60558-079-1.

[109] Roger Keays and Andry Rakotonirainy. Context-oriented programming. In *Proceedings of the 3rd ACM International Workshop on Data Engineering for Wireless and Mobile Access*, pages 9–16. ACM Press, 2003. ISBN 1-58113-767-2. DOI 10.1145/940923.940926.

[110] Gregor Kiczales and Jim Des Rivieres. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991. ISBN 0262111586.

[111] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings of the European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, 1997. ISBN 978-3-540-63089-0. DOI 10.1007/BFb0053381.

[112] Ekkart Kindler. epnk: A generic pnml tool - users and developers guide. Technical Report 3, DTU Informatics, Copenhagen, Denmark, 2011.

[113] Ekkart Kindler and Wil van der Aalst. Liveness, fairness, and recurrence in petri nets. *Inf. Process. Letter*, 70(6):269–27, 1999.

[114] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16:1293–1306, November 1990.

[115] Lars Michael Kristensen and Thomas Mailund. A generalised sweep-line method for safety properties. In *Proceedings of the International Symposium on Formal Methods Europe*, volume 2391 (LNCS) of *FME'02*, pages 549–567. Springer-Verlag, 2002.

[116] Peep Küngas. Petri net reachability checking is polynomial with optimal abstraction hierarchies. In *Proceedings of the 6th international conference on Abstraction, Reformulation and Approximation*, SARA'05, pages 149 – 164, Airth Castle, UK, 2005. Springer-Verlag. ISBN 3-540-27872-9, 978-3-540-27872-6.

[117] Robert Laddaga. Self-adaptive software. Technical Report 98-12, DARPA BAA, 1997.

[118] Robert Laddaga. Self adaptive software problems and projects. In *Proceedings of the Second International IEEE Workshop on Software Evolvability*, SOFTWARE-EVOLVABILITY '06, pages 3–10, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2698-5. DOI 10.1109/SOFTWARE-EVOLVABILITY.2006.10. URL `http://dx.doi.org/10.1109/SOFTWARE-EVOLVABILITY.2006.10`.

[119] Leslie Lamport. Fairness and hyperfairness. *Distributed Computing*, 13(4):239–245, 2000.

[120] Diego Latella, Istvan Majzik, and Mieke Massink. Towards a formal operational semantics of uml statechart diagrams. In *Proceedings of the IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems*, FMOODS'99, pages 465–482, Deventer, The Netherlands, 1999. Kluwer, B.V. ISBN 0-7923-8429-6.

[121] Truong Giang Le, Olivier Hermant, Matthieu Manceny, and Renaud Pawlak. Dynamic adaptation through event reconfiguration. In *Proceedings of the 2011th Confederated international conference on On the move to meaningful internet systems*, OTM'11, pages 637–646, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-25125-2.

[122] Meir M. Lehman and Juan F. Ramil. Software evolution and software evolution processes. *Annals of Software Engineering*, 14:275–309, 2002.

[123] Jean-Christophe Libbrecht and Julien Goffaux. Subjective-C: Enabling context-aware programming on iPhones. Master's thesis, Louvain School of Engineering, Université catholique de Louvain, Belgium, June 2010.

[124] Jens Lincke, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. An open implementation for context-oriented layer composition in contextjs. *Sci. Comput. Program.*, 76(12):1194–1209, December 2011.

[125] Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007. ISBN 3540714367.

[126] Yu Liu and Rene Meier. Resource-aware contracts for addressing feature interaction in dynamic adaptive systems. *Autonomic and Autonomous Systems, International Conference on*, 0:346–350, 2009.

[127] Seng W. Loke. Representing and reasoning with situations for context-aware pervasive computing: a logic programming perspective. *Knowl. Eng. Rev.*, 19 (3):213–233, September 2004.

[128] Irina A. Lomazova. Nested petri nets - a formalism for specification and verification of multi-agent distributed systems. *Fundamenta Informaticae - Special issue on Concurrency specification and programming (CS&P)*, 43(1 – 4):195 – 214, August 2000.

[129] Andoni Lombide Carreton. *Ambient-Oriented Dataflow Programming for Mobile RFID-Enabled Applications*. PhD thesis, Vrije Universiteit Brussel, Pleinlaan 2, Brussels, Belgium, October 2011.

[130] A. Mazurkiewicz. *Compositional semantics of pure place/transition systems*, pages 307 – 330. Springer-Verlag, London, UK, 1988.

[131] Roxana Melinte, Olivia Oanea, Ioana Olga, and Ferucio Laurenţiu Ţiplea. The home marking problem and some related concepts. *Acta Cybernetica*, 15(3): 467–478, 2002.

[132] Stephen J. Mellor, Anthony N. Clark, and Takao Futagami. Guest editors' introduction: Model-driven development. *IEEE Software*, 20:14–18, 2003.

[133] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: a programming language for ajax applications. *SIGPLAN Not.*, 44(10):1–20, October 2009.

[134] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes (parts i and ii). *Information and Computation*, 100(1):1–77, September 1992. ISSN 0890-5401.

[135] Brice Morin, Olivier Barais, Jean-Marc Jezequel, Franck Fleurey, and Arnor Solberg. Modelsrun.time to support dynamic adaptation. *Computer*, 42(10): 44–51, October 2009. ISSN 0018-9162. DOI 10.1109/MC.2009.327. URL `http://dx.doi.org/10.1109/MC.2009.327`.

[136] Stijn Mostinckx, Christophe Scholliers, Eline Philips, Charlotte Herzeel, and Wolfgang De Meuter. Fact spaces: Coordination in the face of disconnection. In *International Conference on Coordination Models and Languages*, volume 4467, pages 268–285, June 2007. ISBN 978-3-540-72793-4. DOI 10.1007/978-3-540-72794-1_15.

[137] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541 – 580, April 1989.

[138] Radu Muschevici, Dave Clarke, and José Proença. Feature petri nets. In Goetz Botterweck, Stan Jarzabek, Tomoji Kishi, Jaejoon Lee, and Steve Livengood, editors, *Proceedings of the 14th International Software Product Line Conference*, volume 2 of *SPLC'10*, Jeju, South Korea, September 2010. Lancaster University.

[139] Julia Padberg. Safety properties in petri net modules. *Journal on Integrated Design and Process Technology*, 8(4):65 – 78, 2004.

[140] Wojciech Penczek and Agata Pólrola. *Advances in Verification of Time Petri Nets and Timed Automata: A Temporal Logic Approach*, volume 20 of *Studies in computational Intelligence*. Springer-Verlag New York, Inc., 2006.

[141] James F. Peters, editor. *Special Issue: Threads in Fuzzy Petri Nets Research*, volume 14. Wiley, August 1999.

[142] James L. Peterson. Petri nets*. In *Computing Surveys*, volume 9, pages 223 – 252. ACM, September 1977.

[143] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, June 1981. ISBN 978-0136619833.

[144] Carl Adam Petri. *Communication with Automata*. PhD thesis, 1962.

[145] Luís Pina and Jo ao P. Cachopo. Atomic dynamic upgrades using software transactional memory. In *International Workshop on Hot Topics in Software Upgrades*, HotSWUp'12, pages 21 – 25, Zurich, Switzerland, June 2012. IEEE. DOI http://dx.doi.org/10.1109/HotSWUp.2012.6226612.

[146] Jorge Pinna Puissant, Ragnhild Van Der Straeten, and Tom Mens. Badger: A regression planner to resolve design model inconsistencies. In *European Conf. Modelling Foundations and Applications (ECMFA)*, volume 7349 of *Lecture Notes in Computer Science*, pages 146–161. Springer, 2012. ISBN 978-3-642-31490-2.

[147] Franck Pommereau. Quickly prototyping petri nets tools with snakes. *Petri net newsletter*, pages 1–18, October 2008.

[148] Thibault Poncelet and Loïc Vigneron. The Phenomenal gem: Putting features as a service on rails. Master's thesis, Université catholique de Louvain, June 2012.

[149] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *Proceedings of the International Conference on Aspect-Oriented Software Development*, pages 141–147. ACM Press, 2002. ISBN 1-58113-469-X. DOI 10.1145/508386.508404.

[150] Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in sat based formal verification. *Journal on Software Tools for Technology Transfer*, 7(2):156 – 173, 2005.

[151] Christian Prehofer. Feature-oriented programming: A new way of object composition. *Concurrency and Computation: Practice and Experience*, 13(5):465–501, April 2001.

[152] Mario Pukall, Christian Kästner, Walter Cazzola, Sebastian Götz, Alexander Grebhahn, Reimar Schröter, and Gunter Saake. Javadaptor—flexible runtime updates of java applications. *Software: Practice and Experience*, 2012.

[153] Damien Rambout. Subjective-C 2.0: Multithreaded context-oriented programmming with Objective-C. Master's thesis, Louvain School of Engineering, Université catholique de Louvain, Belgium, June 2011.

[154] L Recalde, Silva M, Ezpeleta J, and Teruel E. Petri nets and manufacturing systems: An examples-driven tour. In J Desel, Reisig W, and Rozenberg G, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 742–788. Springer-Verlag, 2004.

[155] Klaus Reinhardt. Reachability in petri nets with inhibitor arcs. *Electronic Notes in Theoretical Computer Science*, 223:239–264, 2008.

[156] Wolfgang Reisig. *Petri nets: An introduction*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1982.

[157] Thomas Reps, Mooly Sogiv, and Susan Horwitz. Interprocedural dataflow analysis via graph reachability. TR 94 – 14, Datalogisk Institut - University of Copenhagen, Copenhagen, Denmark, April 1994.

[158] Daniel Salber, Anind K. Dey, and Gregory D. Abowd. The context toolkit: Aiding the development of context-enabled applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 434–441. ACM Press, 1999. ISBN 0201485591. DOI 10.1145/302979.303126.

[159] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2):14:1–14:42, May 2009.

[160] Guido Salvaneschi. Context-oriented programming: A software engineering perspective. *Journal of Systems and Software*, 85(8):1801–1817, August 2012.

[161] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. An analysis of language-level support for self-adaptive software. *Transactions on Autonomous and Adaptive Systems*, page to appear.

[162] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. Javactx: Seamless toolchain integration for context-oriented programming. In *International Workshop on Context-Oriented Programming*, COP'11, July 2011.

[163] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. Contexterlang: Introducing context-oriented programming in the actor model. In *Proceedings of the International Conference on Aspect-Oriented Software Development*, AOSD'12, pages 191–202. ACM Press, 2012. ISBN 978-1-4503-1092-5. DOI 10.1145/2162049.2162072.

[164] Hans Schippers, Michael Haupt, and Robert Hirschfeld. An implementation substrate for languages composing modularized crosscutting concerns. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 1944–1951, New York, NY, USA, 2009. ACM.

[165] Hans Schippers, Tim Molderez, and Dirk Janssens. A graph-based operational semantics for context-oriented programming. In *Proceedings of the 2nd International Workshop on Context-Oriented Programming*, COP '10, pages 1–6. ACM, 2010. ISBN 978-1-4503-0531-0. DOI 10.1145/1930021.1930027.

[166] Gregor Schmidt. Contextr and contextwiki. Master's thesis, Hasso Plattner Institut, Potsdam, Germany, 2008.

[167] Karsten Schmidt. Stubborn sets for standard properties. In *Proceedings of the 20th International Conference on Application and Theory of Petri Nets*, pages 46–65, London, UK, UK, 1999. Springer-Verlag.

[168] Karsten Schmidt. Model-checking with coverability graphs. *Formal Methods in System Design*, 15:239–254, 1999.

[169] Karsten Schmidt. Lola: a low level analyser. In *Proceedings of the 21st international conference on Application and theory of petri nets*, ICATPN'00, pages 465–474, Berlin, Heidelberg, 2000. Springer-Verlag.

[170] Karsten Schmidt. How to calculate symmetries of petri nets. *Acta Informatica*, 36(7):545–590, January 2000.

[171] Karsten Schmidt. Using petri net invariants in state space construction. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, pages 473–488. Springer Berlin / Heidelberg, 2003.

[172] Christian Schubert and Michael Perscheid. Dynamic contract layers for python. Master's thesis, Hasso Plattner Institut, Potsdam, Germany, 2008.

[173] Manuel Silva. B-fairness and structural b-fairness in petri net models of concurrent systems. *Journal of Computer and System Sciences*, 44(3):447–477, June 1992.

[174] Yannis Smaragdakis and Don Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2):215–255, 2002. ISSN 1049-331X. DOI http://doi.acm.org/10.1145/505145.505148.

[175] Carl-Fredrik Sørensen, Maomao Wu, Thirunavukkarasu Sivaharan, Gordon S. Blair, Paul Okanda, Adrian Friday, and Hector Duran-Limon. A context-aware middleware for applications in mobile ad hoc environments. In *Proceedings of*

the 2nd workshop on Middleware for pervasive and ad-hoc computing, MPAC '04, pages 107–110. ACM, 2004. ISBN 1-58113-951-9. DOI 10.1145/1028509.1028510. URL `http://doi.acm.org/10.1145/1028509.1028510`.

[176] George Spanoudakis and Andrea Zisman. Inconsistency management in software engineering: Survey and open research issues. In *in Handbook of Software Engineering and Knowledge Engineering*, pages 329–380. World Scientific, 2001.

[177] Harald Störrle. An evaluation of high-end tools for petri nets. Technical report, Ludwig-Maximilians-Üniversitat München, Bericht 9802, june 1998.

[178] Ragnhild Van Der Straeten. *Inconsistency Management in Model-Driven Engineering*. PhD thesis, Vrije Universiteit Brussel, Pleinlaan 2, Brussels, Belgium, September 2005.

[179] Robert S. Streett and E. Allen Emerson. An automata theoretic decision procedure for the propositional mu-calculus. *Information and Computation*, 81(3): 249 – 264, 1989.

[180] William Strunk and E.B. White. *The Elements of Style*. Longman, fourth edition, 2000. ISBN 0-205-30902-X.

[181] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, AOSD '03, pages 21–29, New York, NY, USA, 2003. ACM. ISBN 1-58113-660-9.

[182] Gabriele Taentzer. Agg: A graph transformation environment for modeling and validation of software. In *AGTIVE*, volume 3062, pages 446–453. Springer, 2003.

[183] Lixin Tao. Shifting paradigms with the application service provider model. *Computer*, 34(10):32–39, October 2001.

[184] Eddy Truyen, Nicolás Cardozo, Stefan Walraven, Jorge Vallejos, Engineer Bainomugisha, Sebastian Günther, Theo D'Hondt, and Wouter Joosen. Context-oriented programming for customizable saas applications. In *Symposium on Applied Computing*, SAC'12, pages 418–425, Trento, Italy, March 2012. ACM. ISBN 978-1-4503-0857-1.

[185] Jorge Vallejos, Sebastián González, Pascal Costanza, Wolfgang De Meuter, Theo D'Hondt, and Kim Mens. Predicated generic functions: Enabling context-dependent method dispatch. In Benoît Baudry and Eric Wohlstadter, editors, *Software Composition*, volume 6144 of *Lecture Notes in Computer Science*, pages 66–81. Springer-Verlag, 2010. ISBN 978-3-642-14045-7. DOI 10.1007/978-3-642-14046-4_5.

[186] Wil van der Aalst. The application of petri nets to workflow management. *Journal of Circuits, Systems, and Computers*, 8(1):21–66, 1998.

[187] Yves Vandewoude. *Dynamically updating component-oriented systems*. PhD thesis, Katholieke Universiteit Leuven, March 2007.

[188] Paulo Verissimo and Antonio Casimiro. Event-driven support of real-time sentient objects. In *Proceedings of the International Workshop on Object-Oriented Real-Time Dependable Systems*, Guadalajara, Mexico, January 2003. IEEE.

[189] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, Simon Helsen, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, 2006. ISBN 978-0-470-02570-3.

[190] Martin von Löwis, Marcus Denker, and Oscar Nierstrasz. Context-oriented programming: Beyond layers. In *Proceedings of the International Conference on Dynamic languages*, pages 143–156, New York, NY, USA, 2007. ACM Press. ISBN 978-1-60558-084-5. DOI 10.1145/1352678.1352688.

[191] Stefan Walraven, Eddy Truyen, and Wouter Joosen. A middleware layer for flexible and cost-efficient multi-tenant applications. In *ACM/IFIP/USENIX 12th International Middleware Conference*, volume 7049 of *Lecture Notes in Computer Science*, pages 370–389. IFIP/Springer, December 2011.

[192] Benjamin Hosain Wasty, Amir Semmo, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. Contextlua: dynamic behavioral variations in computer games. In *Proceedings of the 2nd International Workshop on Context-Oriented Programming*, COP '10, pages 5:1–5:6, New York, NY, USA, June 2010. ACM.

[193] M. Weiser, R. Gold, and J. S. Brown. The origins of ubiquitous computing research at PARC in the late 1980s. *IBM Systems Journal*, 38(4):693–696, 1999. ISSN 0018-8670.

[194] Mark Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):75 – 84, 1993.

[195] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, Brussels, Belgium, January 2001.

[196] Chongyi Yuan, Yu Huang, Wen Zhao, and Xinpeng Li. A study on fairness of place/transition systems to make fairness fairer. *Transactions of the Institute of Messurement and Control*, 33(1):50–58, February 2011.

List of Terms

$A_{\restriction_B}$ Restriction of the set A to the elements of set B, 109
P_c Context places, 124
P_t Temporary places, 124
T_e External transitions, 124
T_i Internal transitions, 124
$[b]$ Equivalence class of method b, 165
Σ Set of labels, 124
Υ Petri net step, 107
Υ Sequence of transition firings in CoPN, 152
$\wedge$ Conjunction dependency relation symbol, 141
$\vee$ Disjunction dependency relation symbol, 231
λ Labeling function, 124
$\longleftrightarrow$ Arc synthesis of arcs $f(p,t)$ and $f(t,p)$, 132
$\mathbb{Z}^*$ Non-negative integers, 106
$\mathcal{B}$ Set of behavioral adaptations, 164
$\mathcal{C}$ Singleton context CoPN, 125, 127
$\mathcal{E}$ Set of enabled internal transitions, 153
$\mathcal{L}$ Set of token colors, 124
$\mathcal{O}$ Ordering of behavioral adaptations, 165
$\mathcal{P}$ Context Petri net, 124
$\mathcal{R}$ A set of context dependency relations, 129
$\mathcal{S}$ A set of singleton CoPNs, 129
$\mathcal{T}$ Stack of fired transitions, 153
$\mathscr{P}$ Set of all CoPNs, 125
$\mathscr{R}$ Set of all context dependency relations, 129
$\mathscr{S}$ Set of all singleton context CoPNs, 127
$\mathscr{T}$ Initial state of the system, 153
$\multimap$ Inhibitor arcs pictogram, 117
ω supremum of $\mathbb{Z}^*$, 110
π_i i^{th} projection function, 154
ρ Priority function, 124
$\tilde{\mathcal{E}}$ Saved state of firing transitions, 153
$\tilde{m}$ Consistent marking multiset of $\mathcal{P}$, 152
$\wp$ Power set symbol, 129
f Flow function, 124
$f_\circ$ Flow function inhibitor arcs, 124
m_0 Initial marking multiset, 124
$*(\cdot,\ldots,\cdot)$ Conjunction dependency DSL operator, 161
$+(\cdot,\ldots,\cdot)$ Disjunction dependency DSL operator, 161
-> Causality dependency DSL operator, 161
--> Suggestion dependency DSL operator, 161
=< Requirement dependency DSL operator, 161
=> Implication dependency DSL operator, 161
>< Exclusion dependency DSL operator, 161
$-\triangleright$ Causality dependency relation pictogram, 88, 134
$\rightarrow$ Conjunction dependency relation pictogram, 141
$-\diamond$ Disjunction dependency relation pictogram, 231
$\square$–$\square$ Exclusion dependency relation pictogram, 88, 133
$-\blacktriangleright$ Implication dependency relation pictogram, 88, 136
$-\blacktriangleleft$ Requirement dependency relation pictogram, 89, 139
$--\triangleright$ Suggestion dependency relation pictogram, 234

Acronyms

ADT	Abstract Data Type
AOP	Aspect-Oriented Programming
API	Application Programming Interface
AST	Abstract Syntax Tree
CPN	Colored Petri nets
CLOS	Common Lisp Object System
CAA	Context-Aware Application
CODA	Context-Oriented Domain Analysis
COP	Context-Oriented Programming
CoPN	context Petri net
DSL	Domain-Specific Language
DASS	Dynamically Adaptive Software Systems
EBNF	Extended Backus-Naur Form
FODA	Feature-Oriented Domain Analysis
FOP	Feature-Oriented Programming
LTL	Linear Temporal Logic
LTS	Labeled Transition Systems
LoLA	Low Level Petri net Analyzer
MOP	Metaobject Protocol
MDE	Model-Driven Engineering
OOP	Object-Oriented Programming
PaaS	Platform-as-a-Service
ePNK	Petri Net Kernel
POI	Points of Interest
SAT	boolean SATisfiability problem
SOA	Service-Oriented Architecture
SaaS	Software-as-a-Service
SPL	Software Product Line
WSDL	Web Services Description Languages