Context Petri Nets
Definition and Manipulation

Nicolás Cardozo[1,2], Sebastián González[1], Kim Mens[1], and Theo D'Hondt[2]

[1] ICTEAM Institute, Université catholique de Louvain
Place Sainte-Barbe 2, 1348 Louvain-la-Neuve, Belgium
{nicolas.cardozo, s.gonzalez, kim.mens}@uclouvain.be
[2] Software Languages Lab, Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussels, Belgium
tjdhondt@vub.ac.be

Abstract. Context-oriented programming languages provide dedicated programming abstractions to define behavioral adaptations and means to recompose them dynamically according to sensed context changes. Additionally, some of these languages have proposed abstractions to explicitly define dependency relations between adaptations. Such dependency relations enable programmers to specify allowed and disallowed interactions between behavioral adaptations at a high abstraction level. In this paper, we explore Petri nets as an underlying formalism to model context-dependent adaptations and their dependencies. Petri nets offer a precise notation and semantics for behavioral adaptations and the dependency relations between them. Even more, Petri nets can be used as an underlying representation to manage dependency relations, and activations and deactivations of behavioral adaptations at runtime. We illustrate the ideas through the context-oriented programming language Subjective-C.

1 Introduction

Current computing platforms consist of highly interconnected computers with access to rich context information. Applications developed with context in mind can leverage the full potential of these platforms by adapting their behavior dynamically according to sensed context changes. To support the development of such applications, the Context-Oriented Programming (COP) paradigm has emerged to allow the definition, composition, and management of context-dependent behavioral adaptations at run time.

In COP languages, the dynamic composition and management of context-dependent adaptations has proven to be a challenging task. Systems must be able to ensure that the addition and removal of behavioral adaptations, respect the expected application behavior. Different approaches have been proposed to ensure such consistency by defining *dependency relations* among adaptations [8,10,14]. These dependencies constrain *adaptation interaction* by conditioning the activation and deactivation of adaptations at a high abstraction level that is well-suited to programmers.

Unfortunately, existing approaches often define such adaptation dependencies and their interaction informally, obscuring their semantics, and making it difficult to discover the subtleties of dependency interactions.

In this paper we argue that Petri nets, and more specifically the use of reactive Petri nets with inhibitor arcs and static priorities, are well suited for the formal expression of application adaptation dynamics. This Petri net model is called context Petri nets (CoPN). High-level adaptation dependencies can be mapped naturally to a corresponding CoPN, in which places represent adaptations, and transitions represent the constraints for activation and deactivation of those adaptations. The CoPN model and formalism thus constitutes a middle ground that bridges the gap between the high-level specification of adaptations and their dependencies as specified by the programmer, and the corresponding implementation that handles the actual activation and deactivation of adaptations.

The proposed formalism contributes to the design and implementation of COP languages and applications, because the definition and interpretation of adaptation dependencies becomes precise and straightforward. The execution flow of the CoPN model naturally corresponds to the activation dynamics of adaptations. This provides a concrete view of the application's state and dynamic evolution at any point in time. The state, interaction and constraints of adaptations are fully expressed within the CoPN, which needs not be complemented with extra information or rules describing its semantics. The verification of activation and deactivation of adaptations is eased because both, the events, and their availability to be triggered are explicitly expressed in the CoPN model. Since all needed information for verification is directly available, the CoPN can even be used at run-time to deal with the adaptation dynamics of the application.

The remainder of this paper is organized as follows. Using a motivating example, Section 2 briefly illustrates the COP paradigm, through the *Subjective-C* language, and the need for expressing dependencies between different context-specific behavioral adaptations. Section 3 provides an overview of how adaptation dependencies are defined by *Subjective-C*. Section 4 introduces basic Petri net concepts, as a lead to Section 5, which shows the CoPN definition and formalization. Section 6 describes how the context Petri net model is introduced and used in *Subjective-C*. Section 7 discusses alternative approaches to model and manage context-dependent adaptations. The paper is rounded off with future work and conclusions in Sections 8 and 9.

2 Context-Oriented Programming

Context-Oriented Programming [9] allows software systems to be modularized into behavioral variations that can be activated and deactivated at run-time. Each variation represents a behavioral adaptation that depends on specific properties of the surrounding environment, such as device battery level, user preferences and geographical location. COP languages provide constructs for

the definition of such adapted behavior, and for its dynamic activation and deactivation according to detected changes. Behavior adaptations are activated dynamically when deemed more appropriate to the new context than the currently active behavior.

Various COP languages have been proposed, either as extensions of existing languages such as CLOS, Smalltalk and Java [17], or as entirely new languages [15]. Some of these refer to behavioral adaptations as *layers* and others as *contexts*, but throughout this paper we stick to the term *adaptation* to denote the general notion of behavioral adaptation in any of its manifestations. Whereas the ideas presented in this paper apply to most COP languages, we illustrate them through *Subjective-C* [14], a COP extension of Objective-C.

2.1 General COP Architecture

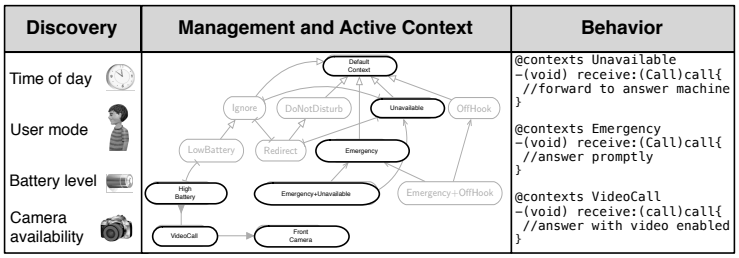


Fig. 1: COP system architecture and interaction.

The envisioned architecture for COP systems that we use as starting point consists of three basic modules[3], illustrated in Fig. 1.

Context discovery module Gathers information about the execution environment of the application, including data such as battery level, user settings, presence of other devices and geographical location. The module then assigns a semantic meaning to the discovered information. For example, it may interpret a battery level below a certain threshold to be a `LowBattery` context.

Context management module Manages the different possible behavioral adaptations taking into account their dependency relations, and orchestrates their composition, according to their appropriateness for the surrounding environment in which the application executes.

Active context module Keeps track of the currently active adaptations at any point during program execution. In the context graph shown in Fig. 1, active adaptations are highlighted in black. Different behavioral adaptations can be simultaneously active.

Application behavior module Defines the behavior for each of the different behavioral adaptations. The third column of Fig. 1 shows a *Subjective-C* example of the definition of different behavioral adaptations when a phone call is

[3] Even though there are four modules, for conciseness, the context management module and active context module are represented together in the figure.

received (forwarding to an answering machine, answering with video enabled, and so on).

2.2 Motivating Example: Phone Call Handling

To illustrate run-time software adaptation, consider the common example of call reception behavior for a mobile phone. The core functionality of the phone consists in advertising incoming calls with a ringtone, and providing a hold mode for simultaneous calls. This core functionality can be enhanced with behavioral adaptations that are deployed according to the context of execution.

The `LowBattery` adaptation is activated whenever the battery level of the phone drops below a predefined level. The activation of `LowBattery` provokes the activation of `Ignore` mode, to save battery by ignoring calls not coming from VIP contacts. The `Ignore` mode can also be activated manually by users. A `DoNotDisturb` adaptation is activated whenever it is detected that the phone should be silent [4] and its advertising method is set to vibrate instead of ringing. This adaptation may be further refined by the `Redirect` adaptation in which calls should be forwarded to another number, for example when in a meeting. The `Unavailable` adaptation forwards calls received at inconvenient times, for example at night, to the phone's answering machine. A `VideoCall` adaptation allows to have video calls on phones with a `FrontCamera` but only when the battery level is sufficiently high. The `Emergency` adaptation allows to receive calls promptly regardless of other applicable situations. Additionally, the combination of some of these adaptations may require special treatment, such as the combination of `Emergency` and `Unavailable`, since one adaptation allows for urgent calls whereas the other transfers all calls to the answering machine. This dedicated behavior can be specified in a combined adaptation.

2.3 Subjective-C

Let us now take a closer look at how this motivating example could be implemented in a COP language like *Subjective-C*. The third column of Fig. 1 already illustrates how to define behavioral adaptations: essentially they are regular Objective-C methods with a special `@contexts` annotation that indicates in presence of which contexts is the method appropriate. Snippet 1 shows the adapted behavior for receiving a call, in the cases of `Unavailability` and `Redirection`.

In this subsection we will focus on the language abstractions needed to manage the definition and consistent interaction of adaptations in *Subjective-C*. A detailed explanation of the different context-oriented language constructs and their implementation can be found in the seminal *Subjective-C* paper [14].

To deal with unexpected or contradicting behavior rising from the combination of adaptations, *Subjective-C* allows to express dependency relations

[4] How the phone would detect that it is in a context where silent mode is preferred is the responsibility of the context discovery module and out of the scope of this paper.

```
@contexts Unavailable
-(void) receive:(Call*) call{
  //Forward to answer machine
}
```

```
@contexts Redirect
-(void) receive:(Call*) call{
  //Redirect to secretary
}
```

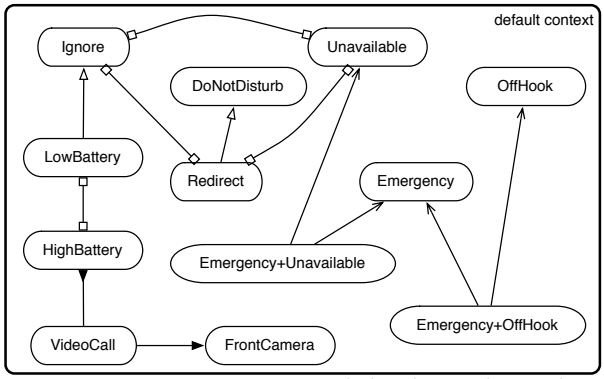
Snippet 1: Behavioral adaptations for the receive method.

```
SCContext *lowBattery = @context(@"LowBattery");
SCContext *ignore = @context(@"Ignore");
[contextManager addWeakInclusionFrom:lowBattery to:ignore];
```

Snippet 2: Weak inclusion dependency relation declaration

between adaptations. Such dependency relations can be defined programmatically by means of dedicated language constructs as shown in Snippet 4. Each of these dependencies, as will be detailed later, imposes constraints on the activation and deactivation of adaptations. The weak inclusion dependency relation between `LowBattery` and `Ignore` exemplified in Snippet 4 causes the `Ignore` adaptation to be activated as a consequence of the activation of `LowBattery`.[5]

A *dependency graph* is used as an intuitive and compact representation of adaptations and their dependencies. To concisely define such adaptation dependencies, *Subjective-C* is complemented with a domain-specific *Context Declaration Language (CDL)*. Fig. 2 shows the dependency graph (left) and its corresponding CDL (right) for the mobile phone example. Both the dependency graph and native *Subjective-C* constructs such as those of Snippet 4 are generated from the declared contexts and dependency relations.




```
LowBattery -> Ignore
VideoCall =< HighBattery
LowBattery >< HighBattery
VideoCall => FrontCamera
Redirect -> DoNot Disturb
Redirect >< Ignore
Redirect >< Unavailable
Ignore >< Unavailable
```

Fig. 2: Mobile phone dependency graph

The rectangle enclosing the dependency graph in Fig. 2, represents the core behavior of the application, to which we refer to as the *default context*. The decorated arcs represent the currently existing types of dependency relations

[5] The inclusion is "weak" in that changes to the target of the dependency, in this case `Ignore`, do not affect the state of the source, `LowBattery`.

```
[receiveCall: phone];  // phone rings
@activate(ignore) //explicit activation of ignore mode
// Ignore COUNT = 1
[receiveCall: phone];  // call ignored
//phone battery is low, triggering @activate(lowBattery)
//request which in turn triggers @activate(ignore)
// LowBattery COUNT = 1 -  Ignore COUNT = 2
[receiveCall: phone];  //call still ignored
@deactivate(ignore); //explicit deactivation of ignore mode
// LowBattery COUNT = 1 -  Ignore COUNT = 1
[receiveCall: phone];  //call still ignored
//phone battery is high, triggering @deactivate(lowBattery)
//which in turn triggers @deactivate(ignore)
// LowBattery COUNT = 0 -  Ignore COUNT = 0
[receiveCall: phone]; // phone rings
```

Snippet 3: Keeping track of active adaptations with activation counters

between adaptations. Edges ending with empty triangles (–▷) represent weak inclusions, full triangles (–►) represent strong inclusions, inverse full triangles (–◄) represent requirement relations, and edges with squares on both sides (□–□) represent exclusions. The meaning of these different types of dependencies will be explained in detail in Section 3.

Fig. 2 also contains simple arrows (→) to represent composition dependencies. Such dependencies denote behavioral adaptations that combine two previously defined adaptations. Intuitively, the composed adaptation is available if and only if all of its components are available. The actual semantics of this composition dependency will be presented in Section 5.

To verify whether it is possible to activate an adaptation,[6] the constraints imposed by the dependencies must be satisfied. Take for example the activation of the `Unavailable` adaptation in Fig. 2. Each of the dependent adaptations must be checked. In order to activate `Unavailable` both `Ignore` and `Redirect` must be inactive. In case the `Emergency` adaptation is already active, the `Emergency`+`Unavailable` adaptation must be activated. This verification process needs to be repeated for each of the adaptations that are implicitly activated as a consequence of activating the `Unavailable` adaptation, and thus may propagate throughout the entire graph.

As illustrated in this verification process, activation of an adaptation may trigger the automatic activation of other adaptations as a consequence of the defined dependency relations. For example, in the weak inclusion dependency defined in Snippet 4, the activation of `LowBattery` will automatically trigger the activation of `Ignore`.

To handle these kind of situations, the notion of *activation counters* was introduced [13]. Activation counters work much like the retain/release mechanism

[6] Throughout the paper we refer to the activation case, although the whole discussion applies to the deactivation case as well.

of memory management systems based on reference counting. Snippet 3 illustrates the idea. Essentially, when an adaptation is activated directly or indirectly, its activation counter is incremented by 1, and when it is deactivated, the activation counter is decremented by 1. Only when the activation counter becomes zero the adaptation is considered inactive. Snippet 3 shows how the activation counter mechanism maintains the expected behavior throughout interleaved (direct or indirect) activation and deactivation of adaptations.

Having introduced the *Subjective-C* language and the different kinds of dependency relations between adaptations, Section 3 takes a closer look at these different kinds of dependency relations.

3 Adaptation Dependencies

This section presents in more detail the different adaptation dependencies originally introduced in *Subjective-C*. The presentation of dependencies given in this section in term of events, predicates and rules already constitutes an improvement that makes more precise the original definition of dependencies in *Subjective-C*. This updated definition of dependencies is the basis from which we develop the formal semantics in terms of context Petri nets in Section 5.

To define the semantics of the different dependencies, two auxiliary events and a predicate are introduced. The events *act(..)* and *deact(..)* express the activation and deactivation of an adaptation, whereas the *isAct(..)* predicate checks whether an adaptation is active. We use the notation $E_1 \Rightarrow E_2$ to express that the event E_1 automatically triggers E_2. The notation $\frac{C}{R}$ expresses the fact that a rule (or event) R can be triggered only when condition C is valid. For example $\frac{\neg isAct(A_1)}{deact(A_2)}$ means that A_2 can be deactivated only when A_1 is inactive. In addition, rules with deactivation events will always assume implicitly adaptations to be deactivated are in fact active, since it does not make sense to deactivate an already inactive adaptation.

For each dependency follows a specification providing, a description of its purpose, notation and intuitive semantics.

Weak inclusion Weak inclusion represents a dependency relation where the activation (deactivation) of the source adaptation automatically triggers the activation (deactivation) of the target adaptation. However, the dependency is weak in the sense that the target adaptation can still be activated or deactivated independently of the source adaptation.

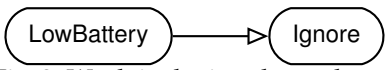


Fig. 3: Weak inclusion dependency

A typical example of a weak inclusion dependency is given in Fig. 3. When the battery level of the phone is low, the `LowBattery` adaptation is turned on. This activation will automatically turn on the `Ignore` adaptation in order to save battery. Similarly, if the phone is charged and the `LowBattery` adaptation is

turned off, the `Ignore` adaptation is not needed anymore, and is also turned off, leaving the default call advertisement behavior. The `LowBattery` adaptation is said to weakly include the `Ignore` adaptation. The inclusion dependency is weak in the sense that the `Ignore` adaptation can be turned on and off independently of the `LowBattery` adaptation. For example, if the `LowBattery` adaptation is activated, `Ignore` will be turned on, but if the user wants to receive all incoming calls, he could turn off the `Ignore` adaptation, regardless of the battery level.
Table 1 shows a rule-based semantics definition of the weak inclusion dependency relation. As explained before, whenever the `LowBattery` adaptation is activated, the `Ignore` adaptation should be activated too. Likewise, deactivation of the `LowBattery` adaptation triggers deactivation of the `Ignore` adaptation. The `Ignore` adaptation may be activated or deactivated independently.

act(`LowBattery`) ⇒ act(`Ignore`)	act(`Ignore`)
deact(`LowBattery`) ⇒ deact(`Ignore`)	deact(`Ignore`)

Table 1: Weak inclusion dependency semantics

Strong inclusion Strong inclusion represents a dependency relation where, similarly to weak inclusions, the activation (deactivation) of the source adaptation automatically triggers the activation (deactivation) of the target adaptation. In this case however, the inclusion is said to be strong because the deactivation of the target adaptation automatically triggers the deactivation of the source adaptation. However, the target adaptation can still be activated independently of the source adaptation. These rules are summarized in Table 2.

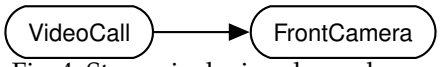


Fig. 4: Strong inclusion dependency

A typical example of a strong inclusion dependency is given in Fig. 4. When a video call request is received, the `VideoCall` adaptation is activated, which will automatically activate `FrontCamera`. When the `VideoCall` adaptation is deactivated, the `FrontCamera` adaptation is deactivated automatically, since the camera was being used for the call. Similarly, if the `FrontCamera` is turned off, the `VideoCall` is automatically turned off and the phone immediately reverts to its default behavior. However, users may activate the `FrontCamera` adaptation independently of `VideoCall`, for example to take photos.

act(`VideoCall`) ⇒ act(`FrontCamera`)	act(`FrontCamera`)
deact(`VideoCall`) ⇒ deact(`FrontCamera`)	
deact(`FrontCamera`) ⇒ deact(`VideoCall`)	$\frac{\neg \text{isAct}(\texttt{VideoCall})}{\text{deact}(\texttt{FrontCamera})}$

Table 2: Strong inclusion dependency relation semantics

Two observations can be made at this point. Firstly, comparing Table 1 and Table 2, the dependency is indeed stronger since the rules are the same except

for the fourth rule (deactivating the target adaptation) which was split in two separate cases: when the source adaptation is not active but the target one is, then the target adaptation can be deactivated freely; if, on the other hand, the two adaptations are active then they are deactivated together[7].
Secondly, shortcomings of the intuitive semantics become apparent. What happens when the adaptations are active multiple times and one of them is deactivated once? Questions like these generate subtle rules that can easily be overlooked, hence motivating the need for a more detailed semantics.

Exclusion The exclusion dependency relation constraints two adaptations so that they cannot be active at the same time. However, both adaptations may be simultaneously inactive.

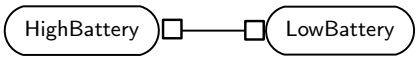


Fig. 5: Exclusion dependency relation

A typical example of a mutual exclusion dependency is given in Fig. 5. The two adaptations `HighBattery` and `LowBattery` are not allowed to be active at the same time, representing the fact that the physical battery cannot simultaneously have a high and a low charge level. If the `HighBattery` adaptation is to be activated, then the `LowBattery` adaptation must be deactivated first, and vice versa. Table 3 shows the intuitive semantics for the exclusion dependency.

$\frac{\neg \text{isAct}(\texttt{LowBattery})}{\text{act}(\texttt{HighBattery})}$	$\frac{\neg \text{isAct}(\texttt{HighBattery})}{\text{act}(\texttt{LowBattery})}$
deact(`HighBattery`)	deact(`LowBattery`)

Table 3: Exclusion dependency semantics

Requirement This dependency represents the situation in which the activation of the source adaptation is possible only if the target adaptation is already active. This restriction implies that the deactivation of the target adaptation automatically triggers the deactivation of the source adaptation.

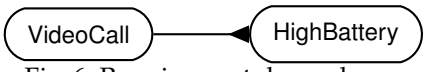


Fig. 6: Requirement dependency

Fig. 6 shows a typical example of the requirement dependency. The `VideoCall` adaptation (source) requires the `HighBattery` adaptation (target) to be active, since processing the video for a call is power consuming. This means that, if the `HighBattery` adaptation is turned off, the `VideoCall` adaptation is automatically deactivated. However, the two adaptations are not tightly related, as activation of the `HighBattery` adaptation is independent of the `VideoCall` one, and

[7] For the rule *deact*(`FrontCamera`) $\Rightarrow$ *deact*(`VideoCall`), remember that we assume all adaptations involved in the rule to be active. Without this assumption the rule would have needed to be written as: $\frac{\textit{isAct}(\texttt{VideoCall}) \wedge \textit{isAct}(\texttt{FrontCamera})}{\textit{deact}(\texttt{FrontCamera}) \Rightarrow \textit{deact}(\texttt{VideoCall})}$

deactivation of the `VideoCall` adaptation is independent of the `HighBattery` adaptation. Table 4 shows the rule-based semantics of the requirement dependency.

$\frac{\text{isAct}_{(\texttt{HighBattery})}}{\text{act}_{(\texttt{VideoCall})}}$	act(`HighBattery`)
$\frac{\neg\text{isAct}_{(\texttt{VideoCall})}}{\text{deact}_{(\texttt{HighBattery})}}$	deact(`VideoCall`)
deact(`HighBattery`) $\Rightarrow$ deact(`VideoCall`)	

Table 4: Requirement dependency semantics

A detailed definition of these dependencies, as well as that of a new composition dependency not introduced previously by *Subjective-C*, is given in Section 5 based on the Petri nets formalism. The formalism is briefly explained in Section 4.

A detailed definition of this dependency, as well as that of a new composition dependency not introduced previously by *Subjective-C*, is given in Section 5 based on the Petri nets formalism, which is briefly explained in Section 4.

4 Petri Nets

Before diving into the detailed definition of adaptation dependencies, this section presents the basic Petri net concepts and an extension that we use for our model. As explained later on, the use of Petri nets to model run-time adaptation in context-oriented programs is driven by the natural mapping of adaptation dependency graphs to Petri nets, as well as the ability of Petri nets to describe the dynamic execution of a system.

Petri nets have been used extensively to describe the information control flow of non-deterministic, concurrent systems. This makes such formalism suitable to model dynamic context changes and interactions between multiple contexts [23]. They provide an abstract means to model system components, and the flow of information between components. Intuitively, a Petri net provides a representation of the states of a system, possible actions over these states, and a specification of when actions can take place.

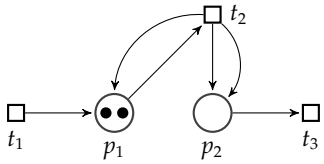
4.1 Basic Petri Nets

Petri nets are directed bipartite graphs, with *places* and *transitions* as disjoint node sets. Formally [23], a Petri net is a quadruple $\mathcal{P} =< P, T, f, m_0 >$ where P is a finite set of places, T is a finite set of transitions, $f : (P \times T) \cup (T \times P) \longrightarrow \mathbb{Z}^+$ is the flow function, and $m_0 : P \longrightarrow \mathbb{Z}^+$ is the initial marking function.

The flow function defines the number of arcs between a place and a transition, and vice versa. There cannot be any arcs between two places or two transitions. A marking assigns *tokens* to places. 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 in every step. The marking function allows for multiple tokens to be assigned to a single place.

Fig. 7 shows an example of a simple Petri net 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 lower part of Fig. 7. The first argument of the function is shown on the rows and the second on the columns. For example, for row t_2 and column p_2, $f(t_2, p_2) = 2$, meaning that there are 2 edges from transition t_2 to place p_2.



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

Fig. 7: Basic Petri net with its flow function f

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 transition of tokens from one place to another after firing the transition. A transition t is enabled (can fire) if *all* its input places $\bullet t = \{p \in P \mid f(p, t) > 0\}$ contain at least $f(p, t)$ tokens. As many as $f(t, p)$ tokens will flow to each output place $t\bullet = \{p \in P \mid f(t, p) > 0\}$. Since many transitions may be enabled at any given time, any of them can be fired, making Petri nets non-deterministic models.

Triggering a transition modifies the marking function m_i to a new marking m_{i+1}. In the example of Fig. 7, firing transition t_2 will yield a new marking m_1, from m_0, where $m_1(p_1) = 2$, $m_1(p_2) = 2$.

Transitions like t_1 with no input places are called sources; they are always enabled. Transitions like t_3 with no output places are called sinks; tokens are removed from the net after their firing.

In the following we present some extensions to the basic Petri net model, needed for the development of our formal model (cf. Section 5).

4.2 Priority Systems

Priority systems [24] provide a means to explicitly express the absence of tokens in a place. Priorities are introduced in Petri nets by adding *zero-testing* or *inhibitor* arcs. Inhibitor arcs, decorated as circle-ended edges (–∘), are given by a flow

function $f_\circ : P \times T \to \{0, 1\}$. There can be maximum one inhibitor arc between a place and a transition.

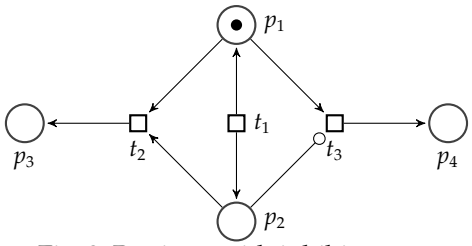


Fig. 8: Petri net with inhibitor arcs

To account for inhibitor arcs the transition firing rules need to be modified. A transition t is said to be enabled if and only if, as before, all of its input places from regular arcs are marked with at least $f(p, t)$ tokens, and *all* of its input places from inhibitor arcs $\{p \in P \mid f_\circ(p, t) = 1\}$ are not marked. Fig. 8 shows an example of a Petri net with one inhibitor arc $f_\circ(p_2, t_3) = 1$. 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 can t_2 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.

4.3 Static Priorities

Static priorities are introduced in Petri nets to fix a firing order over transitions [1,2]. Priorities are given by a function $\rho : T \to \mathbb{Z}^+$ decorating transitions with a weight denoting their firing order. Transitions with a higher priority are fired before transitions with a lower priority. An example of a Petri net with static priorities is shown in Fig. 9. Priorities are shown as small numbers decorating the transitions.

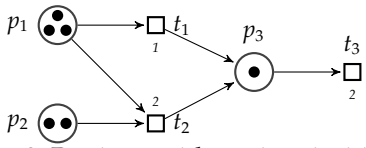


Fig. 9: Petri net with static priorities

The transition firing rules need to be modified to take into account this new restriction. A transition t is enabled if its input places from normal arcs contain at least $f(p, t)$ tokens, its input places from inhibitor arcs are empty, and no other transition in the Petri net with a higher priority is enabled. If two transitions with the same priority are enabled at the same time, they are fired non-deterministically.

In the example of Fig. 9 any of t_2, or t_3 can fire. t_1 cannot fire because it has lower priority than both t_2 and t_3, which are enabled. Only after t_2 has fire twice and t_3 three times (regardless of the order in which they fired), t_1 becomes enabled and can fire.

4.4 Reactive Petri Nets

Reactive Petri nets [12] are introduced to allow the automatic firing of transitions once they are enabled. Such behavior is desired in systems that, as in COP, must *react* automatically as a consequence of changes in the surrounding environment.

Reactive Petri nets split the set of transitions into two sets $T = T_e \cup T_i$, modifying the firing semantics. *External transitions* (T_e) are fired with the regular *may* fire semantics of Petri nets. That is, if a transition is enabled it *may* fire. External transitions can be seen as to fire as a consequence of an external input source. *Internal transitions* (T_i) are fired with a *must* fire semantics. That is, if an internal transition is enabled it must fire. Internal transitions can be seen as to process internal actions of the system.

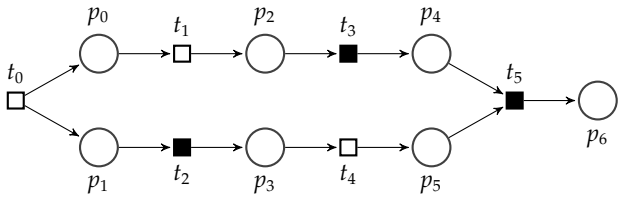


Fig. 10: Reactive Petri net

Fig. 10 shows an example of a reactive Petri net. In the figure, external transitions are represented in white, and internal transitions are represented in black. Transitions t_2, t_3 and t_5 are fired as soon as they become enabled.

Reactive Petri nets introduce the notion of *net stability*. A reactive Petri net is said to be stable if none of its internal transitions is enabled. From this condition it can be seen that in order to have an stable reactive Petri net, all enabled internal transitions must fire before any external transition does. A formal definition of net stability and the equivalence between reactive Petri nets and simple Petri nets is provided in the seminal work on reactive Petri nets [12].

5 Context Petri Nets Runtime Model and Semantics of COP Systems

Having introduced Petri nets, this section presents our formalism for the definition and run-time representation of context-dependent adaptations and their dependencies. The section focuses on the specification of the CoPN formalism and its precise semantics to define COP systems, in particular adaptations, dependency relations and the composition of different adaptations.

The CoPN model corresponding to the dependency graph for the mobile phone application of Fig. 2 can become quite complex as the number of adaptations, and the dependency relations between them increases. The model is rather intended to serve as a precise runtime model that unambiguously specifies the

underlying semantics of the dependency graph of Fig. 2, and to represent the current state of the system, and the allowed activations and deactivations at any point during the system execution.

The remainder of this section gives a formalization for CoPNs. A discussion of the benefits of using CoPN as a formal notation and eventually as an execution model follows in Section 7.

5.1 Context Petri Nets definition

This section presents the formal definition of CoPN and maps its elements onto the different COP concepts.[8]

Definition 1. *A context Petri net is defined as a reactive Petri net, with inhibitor arcs and static priorities* $\mathcal{P} = < P, T, f, f_\circ, \rho, m_0 >$. *Additionally we differentiate between two disjoint sets of places,* context *and* temporary *places, such that* $P = P_c \cup P_t$.

Definition 2. *An adaptation is a particular CoPN defined by the structure in Fig. 11 for which* P_c *is a singleton set. The set of single context CoPNs is denoted as* $\mathcal{S}$. *Hence a particular context is sub-indexed by the name of the context* $C_{\texttt{Ignore}}$, *where* $C_{\texttt{Ignore}} \in \mathcal{S}$.

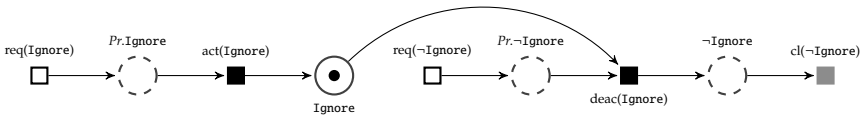


Fig. 11: CoPN $C_{\texttt{Ignore}}$ for a single context `Ignore`

Places in CoPN are used to capture the state of adaptations. An adaptation is defined in terms of four places as depicted in Fig. 11.

- *Context places* P_c (solid border circles in Fig. 11) are used to represent the different adaptations in the system.
- *Temporary places* P_t (dashed border circles in Fig. 11) are introduced to express temporary (preparatory and cleanup) states for an adaptation, easing the consistency verification and composition processes. The temporary places of a CoPN are used for the processing of: a request to activate an adaptation, a request to deactivate an adaptation, or to flag an adaptation is already deactivated. The usefulness of temporary places is evidenced for the composition of CoPNs which is explained in Section 6.1.

Activation and deactivation of an adaptation does not occur immediately, but needs to be requested first and processed carefully, since the request may be denied if the activation or deactivation would violate constraints imposed by other adaptations.

[8] A full implementation of the CoPN runtime model for the context-oriented language Subjective-C is available for download at: `http://released.info.ucl.ac.be/Tools/Context-PetriNets`.

Given that adaptations can be activated multiple times, the flag place ensures that adaptations are deactivated only once per deactivation request. For example, if an adaptation reifies a service that is acquired from multiple sources, and one of the service providers is disconnected but the other ones remain active.
Given an adaptation `Ignore`, as in Fig. 11, its corresponding CoPN $C_{\texttt{Ignore}}$, is realized by: $P_c = \{\texttt{Ignore}\}$ and $P_t = \{Pr.\texttt{Ignore}, Pr.\neg\texttt{Ignore}, \neg\texttt{A}\}$ with exactly 3 places, representing respectively, preparation for activation, preparation for deactivation, and the already deactivated flag.[9]
Transitions represent actions that can be taken on the state of a system. In the case of CoPN, these actions correspond to adaptation activations and deactivations. Transitions are divided into three mutually disjoint sets: T_e, T_i and T_c.

- *External transitions* T_e (white squares in Fig. 11) are used to *request* an adaptation activation or deactivation in response to some change in the surrounding environment. Their priority is given by: $\rho(t_e) = 0, \forall t_e \in T_e$
- *Internal transitions* T_i (black squares in Fig. 11) deal with the constrains imposed by other adaptations, as we will see in Section 3. Internal transitions trigger the actual activation or deactivation of adaptations. Their priority is given by: $\rho(t_i) = 2, \forall t_i \in T_i$
- *Internal-cleaning transitions* T_c (gray squares in Fig. 11) are a particular kind of internal transitions which are used to clean the already deactivated flag ($\neg\texttt{A}$), after all other internal transitions have been fired. Their priority is given by: $\rho(t_c) = 1, \forall t_c \in T_c$

Fig. 11 shows the priority for each of the transitions. In the remainder of this paper priorities will be omitted from figures, as these can be deduced from the color of transitions.
Tokens represent context activations as they reside in places. Depending on in which place a token is, this represents the state of an adaptations, active or inactive. In Fig. 11 the context `Ignore` is active if the place labeled `Ignore` contains a token, preparing for activation if place $Pr.$`Ignore` contains a token, preparing for deactivation if place $Pr.\neg$`Ignore` contains a token, and already deactivated if place $\neg$`Ignore` contains a token.
Inhibitor arcs provide the possibility to verify the absence of tokens in a place. Inhibitors are used to model dependency relations, for example to express that a context can only be activated if some other context is not active.

Generally, a COP system is composed of multiple adaptations (e.g. instances of Fig. 11). A particular adaptation can depend on many other adaptations. First, we formalize different possible dependency relations between adaptations. The definition of the composition operator for CoPN is given in Section 6.1.

5.2 Mapping Dependencies to Petri Nets

Using the mapping scheme presented in Section 5.1, this section presents the corresponding CoPN definition for each of the *dependency relations*. These def-

[9] In CoPNs, labels serve only as a visual decoration to identify places and transitions. Labels have no semantic purpose in the model.

initions are based on the mapping given in Section 5.1, and use the examples given in Section 3.

Taking advantage of the fine grained definition of adaptations previously given, dependency relations can be defined in terms of CoPNs. Dependency relations are defined as a set of constraints describing the interaction between two adaptations. Such constraints are expressed as clauses that must be satisfied by the CoPN.

Definition 3 (Relation constraints). *A constraint on a CoPN* $\mathcal{P} =< P, T, f, f_\circ, \rho, m_0 >$ *is defined as a clause of the form:*

$$Q\, t \in T \text{ such that } B_1(t) \; : \; B_2(t)$$

where Q is a quantifier over the transitions T, B_1 *is a condition over such transitions and* B_2 *is a condition over the flow functions* f *or* $f_\circ$ *that must follow whenever condition* B_1 *holds.*

Take as example the following clauses used to describe part of the CoPN in Fig. 11: $\forall t \in T$ then $\bullet t \neq \phi \vee t\bullet \neq \phi$ (all transitions have predecessors or successors) and $\exists t \in T$ such that $t\bullet = \phi$ (There is a transition that has no successors).

Definition 4 (Dependency relation). *Given two adaptations* $C_1, C_2 \in \mathcal{S}$*, a dependency relation* $R(C_1, C_2)$ *between the two adaptations, is defined as a CoPN* $\mathcal{P}$ *where* $C_1, C_2 \in \mathcal{P}$ *and* $\mathcal{P}$ *satisfies all constraints in the set* $\mathscr{C}_R$ *of constraints for the relation. The set of dependency relations is denoted as* $\mathcal{R}$*.*

Definition 5 (Satisfiability). *We say that a CoPN* $\mathcal{P}$ *satisfies a set of constraints* $\mathscr{C}$*, denoted as* $\mathcal{P} \models \mathscr{C}$*, if and only if* $\forall c \in \mathscr{C}$ *the transitions t in* $\mathcal{P}$ *validate the constraint.*

Satisfiability in CoPN is verified programmatically by going over all $c \in \mathscr{C}$ and verifying if the constraint is valid for the transitions in the CoPN. As it will be seen in Section 6.1, this process takes place every time adaptations are added to the system by means of composition.

Currently, CoPN supports the 4 dependency relations described in Section 3, however, other relations could be defined in a similar fashion. In the following definitions the constraints can be visually identified by the arcs going from one adaptation to another, and the transitions that lie in between them. The set $\mathscr{C}$ of constraints that must be fulfilled and the corresponding CoPN visual representation.

The visual representation given in the following definitions contains double arcs. These are not a new kind of Petri net element, they are just a visual synthesis to make the model less cluttered. A double arc between a place p and a transition t is the synthesis of the arcs (p, t) and (t, p) in f.

Each dependency relation is presented with an intuitive example showing when such dependency relation could be used, and the way the adaptations interact when they are activated. Dependency relations between two adaptations are usually defined based on the domain information of the application. Their

interaction whenever an adaptation is activated or deactivated in CoPN (i.e. the flow of tokens), is explained in Section 5.3.

The CoPN defined by each of the dependency relations is of the form $\mathcal{P} =< P, T, f, f_\circ, \rho, m_0 >$.

Weak inclusion (L–▷I) The conditions that must be satisfied by a weak inclusion are:

$$\begin{aligned}
\mathscr{C}_W : & \exists t \in T \text{ such that } L, Pr.\neg I \in {\bullet}t \text{ and } (I,t) \in f_\circ \\
& \forall t \in T \text{ such that } L \in t\bullet \text{ and } I \notin {\bullet}t \text{ then } (t, Pr.I) \in f \\
& \forall t \in T \text{ such that } L \in {\bullet}t \text{ and } (I,t) \notin f_\circ \text{ then } (t, Pr.\neg I) \in f \\
& \forall t \in T \text{ such that } L \in {\bullet}t \text{ and } (I,t) \notin f_\circ \text{ then } (I,t), (t,I) \in f
\end{aligned}$$

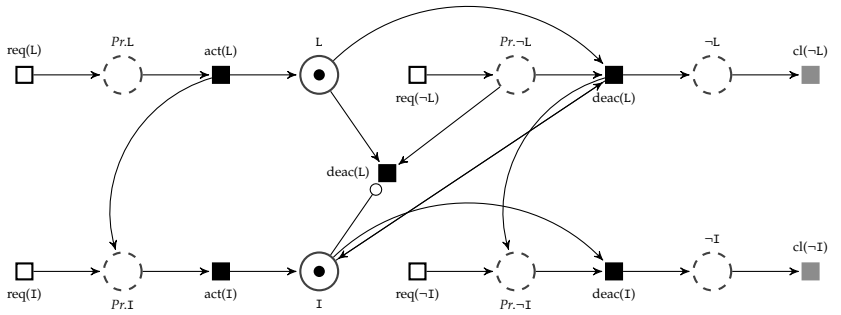


Fig. 12: Weak inclusion dependency relation CoPN definition

Fig. 12 shows the corresponding CoPN for a weak inclusion. Triggering the *act(L)* transition, activates the `LowBattery (L)` adaptation and request the activation for the `Ignore (I)` one. There are two possible transitions to deactivate the `LowBattery` adaptation because, as suggested in the intuitive semantics of Table 1, the deactivation of `LowBattery` must trigger the deactivation of `Ignore`. This can be seen by the rightmost transition labeled *deac*(`L`). However, the `Ignore` adaptation may be freely activated and deactivated. Therefore, it should be possible to deactivate the `LowBattery` adaptation (leftmost *deac*(`L`) transition) even when `Ignore` is inactive (does not contain any tokens). The inhibitor arc accounts for the case in which `Ignore` is already inactive. Such cases expressed with inhibitor arcs are easy to miss when expressing the semantics informally in words or in rules using predicates, as was done in Section 3. The advantage of expressing the semantics formally in terms of CoPNs is the explicit statement of all possible cases in a concise way. Having specified the semantics more formally, it is apparent that there is a rule missing in Table 1 to cover the deactivation of `LowBattery` when `Ignore` is inactive, $\frac{\neg \text{isAct}(\texttt{Ignore})}{\text{deact}(\texttt{LowBattery})}$.

Strong inclusion (V–► F) The conditions that must be satisfied by a strong inclusion are:

$$\begin{aligned}
\mathscr{C}_S : &\exists t \in T \text{ such that } (Pr.\neg V, t), (t, \neg V), (\neg V, t) \in f \\
&\exists t \in T \text{ such that } (Pr.F, t), (t, \neg F), (\neg F, t) \in f \\
&\exists t \in T \text{ such that } F, Pr.\neg F \in {\bullet}t \text{ and } (V, t) \in f_\circ \\
&\forall t \in T \text{ such that } \neg V \in t\,{\bullet} \text{ and } \neg V \notin {\bullet}t \text{ then } (\neg V, t) \in f_\circ \\
&\forall t \in T \text{ such that } \neg F \in t\,{\bullet} \text{ and } \neg F \notin {\bullet}t \text{ then } (\neg F, t) \in f_\circ \\
&\forall t \in T \text{ such that } V \in t\,{\bullet} \text{ and } V \notin {\bullet}t \text{ then } (t, Pr.F) \in f \\
&\forall t \in T \text{ such that } V \in {\bullet}t \text{ and } V \notin t\,{\bullet} \text{ then } (t, Pr.\neg F) \in f \\
&\forall t \in T \text{ such that } F \in {\bullet}t \text{ and } (V, t) \notin f_\circ \text{ then } (t, Pr.\neg V) \in f \\
&\forall t \in T \text{ such that } F \in {\bullet}t \text{ and } (V, t) \notin f_\circ \text{ then } (V, t), (t, V) \in f
\end{aligned}$$

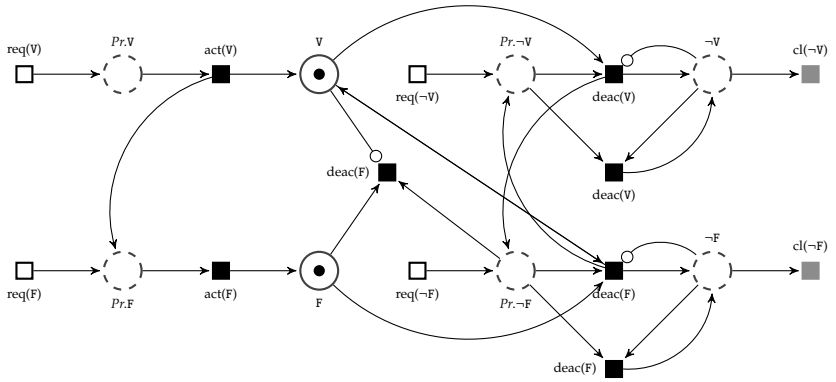


Fig. 13: Strong inclusion dependency relation CoPN definition

The strong inclusion corresponding CoPN is shown in Fig. 13. The difference with the weak inclusion comes from the deactivation of the `FrontCamera (F)` adaptation Now, the deactivation of the `FrontCamera` requests the deactivation of the `VideoCall`. Additionally, the `FrontCamera` may be activated and deactivated independently from the `VideoCall` adaptation. An inhibitor is placed to allow deactivation of the `FrontCamera` when the `VideoCall` adaptation has not been activated.

Exclusion (L □–□ H) The conditions that must be satisfied by an exclusion are:

$$\begin{aligned}
\mathscr{C}_E : &\forall t \in T \text{ such that } L \in t\,{\bullet} \text{ then } (H, t) \in f_\circ \\
&\forall t \in T \text{ such that } H \in t\,{\bullet} \text{ then } (L, t) \in f_\circ
\end{aligned}$$

The exclusion CoPN is shown in Fig. 14. Here, the activation of the two adaptations `HighBattery (H)` and `LowBattery (L)` are restricted by inhibitor arcs coming from the other adaptation. The activations are only enabled if the other place is not marked.

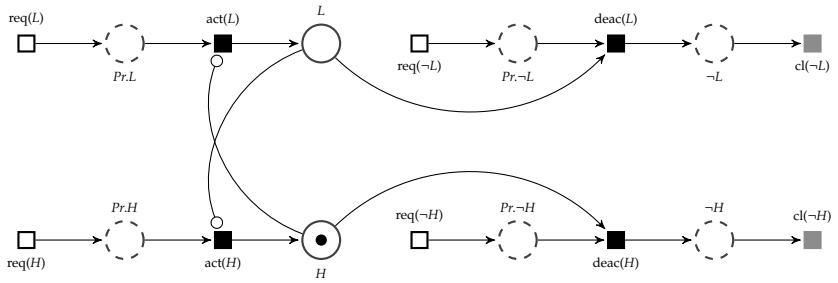


Fig. 14: Exclusion dependency relation CoPN definition

Requirement (V−◀H) The conditions that must be satisfied by a requirement are:

$$
\begin{aligned}
\mathscr{C}_R : \; & \exists t \in T \text{ such that } (Pr.\neg V, t), (t, \neg V), (\neg V, t) \in f \\
& \exists t \in T \text{ such that } H, Pr.\neg H \in {\bullet t} \text{ and } (V, t) \in f_\circ \\
& \exists t \in T \text{ such that } V, Pr.\neg V \in {\bullet t}, \; (H, t) \in f_\circ \\
& \quad \text{and } (\neg V, t), (t, \neg V), (t, Pr.\neg V) \in f \\
& \forall t \in T \text{ such that } \neg V \in t\bullet \text{ and } \neg V \notin {\bullet t} \text{ then } (\neg V, t) \in f_\circ \\
& \forall t \in T \text{ such that } V \in t\bullet \text{ and } V \notin {\bullet t} \text{ then } (H, t), (t, H) \in f \\
& \forall t \in T \text{ such that } H \in {\bullet t} \text{ and } V \notin {\bullet t} \text{ then } (t, Pr.\neg V) \in f
\end{aligned}
$$

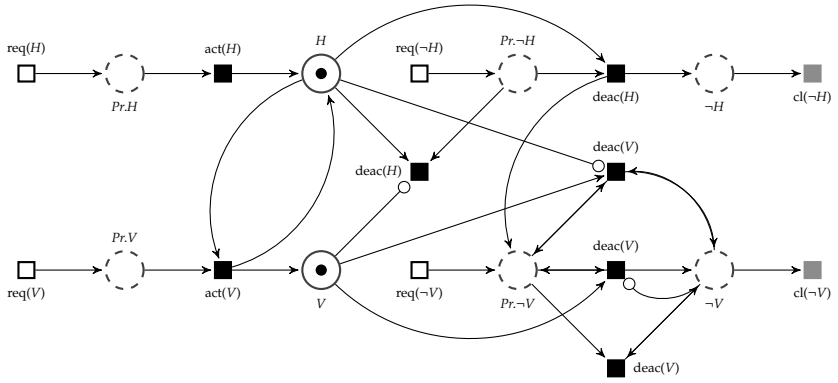


Fig. 15: Requirement dependency relation CoPN definition

The leftmost transitions of the diagram show the independent deactivation and activation of the `HighBattery (H)` adaptation. The inhibitor from the `VideoCall (V)` adaptation to the *deact(H)* is required for the deactivation of the `HighBattery` adaptation when `VideoCall` is inactive. If `VideoCall` would be active, then deactivation of `HighBattery` would also deactivate the `VideoCall` adaptation (rightmost bottom transition). Note that the activation of `VideoCall` is not a source transition, but it has the `HighBattery` place as an input, precisely to express the condition that the `VideoCall` requires `HighBattery`. The transition has an arc going back to the `HighBattery` place, because otherwise the activation

count for `HighBattery` would decrease when activating `VideoCall`. Finally, the deactivation of `VideoCall` can take place independently.

5.3 Adaptation (de)Activation Semantics

Adaptations can be dynamically activated and deactivated as a consequence of changes in the system's execution environment. CoPN semantics is used to ensure a consistent system behavior when adaptations are activated (via the constraints imposed by the dependency relations). A consistent state is always ensured during the execution of the system when using CoPN.

Definition 6. *A CoPN $\mathcal{P}$ is said to be in a* consistent state *if, after all internal transitions have fired, there is no temporary place which is marked.*

The state in a CoPN can only be modified by means of an adaptation activation or deactivation. Whenever any of these actions is triggered in the system for a particular adaptation, the corresponding external transition is fired in the CoPN. More specifically, when adaptation `Ignore` is activated, the transition labeled *req*(`Ignore`) is fired; when adaptation `Ignore` is deactivated, the transition *req*(¬`Ignore`) is fired.

Consider the state of a COP system as a triplet $< \mathcal{P}, \Sigma, m >$ given by its CoPN $\mathcal{P}$, a queue Σ of internal transitions to be fired, and the *current marking* of the system m. Two auxiliary predicate functions are used to describe the activation dynamics of the system. Predicate `marked(p)` tells if place p is marked. Predicate `enabled(t)` tells if a transition t is enabled. The function `action(`*`context-name`*`)` is used to represent a request for an adaptation (de)activation given its name. The function `process(`$\cdot$`)` is used to process the first element of the queue Σ (process a transition t, or to check if the queue is empty). Adaptation activation and deactivation dynamics are expressed by the following rules.

EXTERNAL TRANSITION FIRING, `action`(*context-name*):

$$\frac{m' = m_0 \cup \{p \in P \,|\, \texttt{marked}(p)\},\ \Sigma = \phi}{< \mathcal{P}, \Sigma, m_0 > \rightarrow < [m_0/m']\mathcal{P}, \Sigma \triangleright \{t \in T | \texttt{enabled}(t)\}, m_0 >} \tag{1}$$

Actions are evaluated only when the queue Σ is empty. After the external transition associated with the context is fired, the marking of the CoPN is modified, possibly enabling internal transitions. Such transitions are appended ($\triangleright$) to the end of the queue.

INTERNAL TRANSITION FIRING, `process`(t):

$$\frac{m \in \mathcal{P},\ m' = m \cup \{p \in P \,|\, \texttt{marked}(p)\},\ \Sigma \neq \phi,\ \Sigma' = \Sigma \setminus \{t\}}{< \mathcal{P}, \Sigma, m_0 > \rightarrow < [m/m']\mathcal{P}, \Sigma' \triangleright \{t' \in T \,|\, \texttt{enabled}(t')\}, m_0 >} \tag{2}$$

If the queue Σ is not empty, processing one of its elements i.e., firing the internal transition at the beginning of the queue, yields a new marking of the CoPN. The new marking possibly enables some internal transitions. Such transitions are appended to the end of the queue.

Evaluation termination, process(·):

$$\frac{m \in \mathcal{P},\ \{p \in P_t \,|\, \texttt{marked}(p)\} \neq \phi}{< \mathcal{P}, \phi, m_0 > \rightarrow < [m/m_0]\mathcal{P}, \phi, m_0 >} \tag{3}$$

$$\frac{m \in \mathcal{P},\ \{p \in P_t \,|\, \texttt{marked}(p)\} = \phi}{< \mathcal{P}, \phi, m_0 > \rightarrow < \mathcal{P}, \phi, [m_0/m] >} \tag{4}$$

If there are no internal transitions in the queue Σ to be processed ($\Sigma = \phi$), two outcomes are possible. The first case, when there are temporary places marked in the CoPN, in which all changes to the CoPN are reverted by restating its initial marking. The second case, when none of the temporary places is marked, in which the current marking of the system is replaced by the new marking of the CoPN.

Theorem 1. *Let $\mathcal{P}$ be a CoPN in a consistent state. Any activation or deactivation action σ, of an adaptation $\mathcal{C}$ in $\mathcal{P}$ leaves the system in a consistent state.*

Proof. Firing of σ modifies the marking m of $\mathcal{P}$. Using reduction Reduction rule (1), we know that an external transition firing always marks at least one temporary place. When temporary places are marked we have two cases:
If no internal transition is enabled after the external transition firing, that is $\Sigma = \phi$, Reduction rule (3) is applied. This sets back $\mathcal{P}$ to its original marking m. As no change was produced in $\mathcal{P}$ by action σ. We know by hypothesis that $\mathcal{P}$ is in a consistent state.
If on the contrary, there are internal transitions to be fired, Reduction rule (2) can be applied as many times as needed until the queue Σ is empty. Reduction rule (2) modifies marking of $\mathcal{P}$ to a marking m'. At this point, one of the two reduction rules (3) or (4) can by applied.
case 1: *There is a marked temporary place.*
This case follows similarly as when there are no internal transition to fired, retrieving the CoPN to its original marking m and to a consistent state.
case 2: *No temporary place is marked.*
Applying Reduction rule (4) the marking of $\mathcal{P}$ is updated to marking m'. As there are no temporary places marked in $\mathcal{P}$. Then by Definition 6, $\mathcal{P}$ is in a consistent state. □

Note from Theorem 1 that if a (de)activation σ leads to an inconsistent state —that is, it leads to Reduction rule (3), then, action σ is oblivious to the system and the CoPN is set back to its initial state. Whenever an adaptation activation or deactivation is disregarded because it leads to an inconsistent state, the cause of the inconsistency is signaled to the user.

Example 1. To demonstrate the dynamics of adaptation activation and deactivation in CoPN, consider the sequence

$$\sigma = \{\texttt{activate}(Be), \texttt{activate}(Br), \texttt{deactivate}(Be)\}$$

for the CoPN shown in Fig. 4.

Once the `@activate(Be)` and `@activate(Br)` have been executed, the marking m of the CoPN is $m(Br) = 1$ and $m(Be) = 2$ as illustrated in Fig. 4. The call to `@deactivate(Be)` triggers the enabled external transition req(¬Be) yielding a new marking m_1 where $m_1(Be) = 2, m_1(Br) = 1$ and $m_1(Pr.\neg Be) = 1$. For this marking the only enabled internal transition is the deactivation transition *deac*(*Be*) between places $Pr.\neg Be$ and $\neg Be$. Triggering of such transition (since it must happen) yields a marking m_2 where $m_2(Pr.\neg Be) = 0$, $m_2(Be) = 1$, $m_2(\neg Be) = 1$, $m_2(Pr.\neg Br) = 1$, and $m_2(Br) = 1$. Note that the only enabled transitions are *cl*(¬*Be*), and the *deac*(*Br*) between $Pr.\neg Br$ and $\neg Br$. Transition *cl*(¬*Be*) cannot be fired just yet, since it is enabled but it has a lower priority than the other enabled transitions —the latter must fire first. Firing of *deac*(*Br*) yields the marking m_3 where $m_3(\neg Br) = 1$, $m_3(Pr.\neg Be) = 1$, $m_3(Be) = 1$ and $m_3(\neg Be) = 1$. Now the enabled internal transitions are the leftmost *deac*(*Be*), *cl*(¬*Br*) and *cl*(¬*Be*). Firing *deac*(*Be*) does not enable any other transition since it is a sink transition. The yielded marking is m_4 where $m_4(Be) = 1$, $m_4(\neg Br) = 1$ and $m_4(\neg Be) = 1$. Transitions *cl*(¬*Br*) and *cl*(¬*Be*) have the same priority, and the order in which they fire yields the same result. Once they have both fire none of the internal transitions is enabled and hence the CoPN has reached a consistent state with final marking $m_5(Be) = 1$.

6 Context Petri Net Implementation

This section explores the implementation of the underlying Petri net model to manage activation and deactivation of behavioral adaptations. The implementation provides an extensible library supporting Petri nets with inhibitor arcs in *Objective-C*. The library takes inspiration from the ideas of the Petri Net Kernel,[10] and the SNAKES toolkit [25] and serves as the runtime execution model for the management of context-aware applications in *Subjective-C*.

Adaptation declaration ::= '**@context**('*context-name* [,*bound*]')'
Adaptation activation ::= '**@activate**('*context-name*')'
Adaptation deactivation ::= '**@deactivate**('*context-name*')'
Dependency relations declaration ::=
'['(addExclusionFrom:to: | addStrongInclusionFrom:to:
addRequirementTo:of: | addWeakInclusionFrom:to:)
(*context-name*, *context-name*) ']'
bound :: = NUMBER

Table 5: *Subjective-C* method syntax to interact with CoPNs

Table 5 shows the language constructs for the creation and execution of CoPNs. A *adaptation declaration* automatically generates an adaptation structure as that of Fig. 11. The maximum number of times an adaptation can be activated

[10] See `http://www2.informatik.hu-berlin.de/top/pnk/`

```
1 SCContext *lb = @context(@"LowBattery");
2 SCContext *hb = @context(@"HighBattery");
3 [contextManager addExclusionFrom: lb to: hb];
4 @activate(LowBattery);
5 @activate(HighBattery);
```

Snippet 4: Exclusion dependency declaration

can be defined by a *bound*. Contexts *activation* and *deactivation*, generate the firing of the external transitions associated with and activation and deactivation, respectively, *req*(`Ignore`) and *req*(¬`Ignore`) in Fig. 11. Finally, a *dependency relation declaration* specifies the different interactions between two adaptations. Examples of these are given in Section 3.

Behavioral adaptations are defined in Subjective-C as annotated methods. Snippet 1 shows the definition of two behavioral adaptations of the `receive` method, defined respectively for the `Ignore` and `Redirect` adaptations.

Snippet 4 shows the definition for the `LowBattery` and `HighBattery` adaptations of the mobile phone. Lines 1 and 2 generate a CoPN as the one in Fig. 11. The exclusion dependency defined between the two adaptations in Line 3 yields the CoPN shown in Fig. 5. Line 4 is the activation of the `LowBattery` adaptation which (when successful) installs the behavior adaptations associated to it, the `receive` method for the `Ignore` adaptation in Snippet 1. This adaptation activation retrieves the trace of internal transitions fired to the user. Due to the `LowBattery` adaptation, being active, activation of the `HighBattery` adaptation in Line 5, is denied. The cause of the problem is retrieved to the user.

6.1 Composing Context Petri Nets

In the general case, composing Petri nets is a challenging task [5, Chapter 4]. However, for CoPN we take an algorithmic approach that composes different existing dependencies into a unified Petri net.

A COP system generally comprises multiple adaptations. A particular adaptation may have dependency relations with many other adaptations. In order to create the underlying CoPN for the complete system, all of the adaptations in the system and their dependency relations must be composed. We present the composition operator for CoPNs, which given two CoPNs, generates another one.

Definition 7. *A COP system* $\mathfrak{C}$ *consisting of adaptations* $C_1, \ldots, C_n$ *is defined as a CoPN* $\mathcal{P} = < P, T, f, f_\circ, \rho, m_0 >$ *obtained from the composition of all the adaptations defined in the system, and by possibly adding extra transitions and arcs.*

Definition 8. *The composition operator* $\circ : \mathcal{S} \times \mathcal{S} \rightarrow \mathfrak{C}$ *is defined between two adaptations* $C_1, C_2 \in \mathcal{S}$ *as* $\circ(C_1, C_2) \mapsto \mathcal{P}$*, where* $C_1 =< P_1, T_1, f_1, f_{\circ_1}, \rho_1, m_{01} >$ *and* $C_2 =< P_2, T_2, f_2, f_{\circ_2}, \rho_2, m_{02} >$ *are CoPNs, and* $\mathcal{P} =< P, T, f, f_\circ, \rho, m_0 >$*.* $\mathcal{P}$ *is the combination of these CoPNs by the union of their*

places and transitions, whenever these correspond to each other. Two places (transitions) are said to correspond if their labels, inputs and outputs are the same. $\mathcal{P} = \circ(C_1, C_2)$ *is such that:* $P = P_1 \cup P_2$, $T = T_1 \cup T_2$, $m_0 = m_{01} \cup m_{02}$, $\rho = \rho_1 \cup \rho_2$ *and*

$$f(x,y) = \begin{cases} f_1(x,y) & \text{if } x,y \in C_1 \\ f_2(x,y) & \text{if } x,y \in C_2 \end{cases}$$

$$f_\circ(p,t) = \begin{cases} f_{\circ 1}(p,t) & \text{if } p \in P_1 \text{ and } t \in T_1 \\ f_{\circ 2}(p,t) & \text{if } p \in P_2 \text{ and } t \in T_2 \end{cases}$$

Note that whenever two adaptations are composed, the composition yields one of two cases. The first case is when the two adaptations are different, hence the composition yields a CoPN where there is no arc connecting the adaptations. The second case, is when the two adaptations are the same. In this case the composition yields a CoPN for one of the adaptations.

A more interesting case is when more complex CoPNs are composed. We first consider the case of composing two CoPN, each representing a dependency relation. In such a case, the resulting CoPN must comply to the constraints imposed by both dependency relations.

The definition of the composition operator is extended to compose dependency relations. That is, the operator now takes two of dependency relations as arguments instead of a pair of adaptations. Two dependency relations may be composed if and only if they share a common adaptation. If they do not share an adaptation the composition is trivial.

Definition 9. *The composition operator* $\circ : \mathcal{R} \times \mathcal{R} \to \mathfrak{C}$ *is defined for two dependency relations* $R_1(C_1, C)$ *and* $R_2(C, C_2)$ *where* $\circ(R_1, R_2) \mapsto \mathcal{P}$*. The composition of two dependency relations takes place in two steps: 1. First common adaptations are composed as in Definition 8 (if any) 2. Ensuring that* $\mathcal{P}$ *satisfies the constraints imposed by both relations* $\mathcal{P} \models \mathscr{C}_{R_1} \wedge \mathscr{C}_{R_2}$ *as explained in Section 3.*

Note that the composition operator is defined for any two dependency relations. The order in which the adaptations appear in the relation (source or target) is not important for the composition.

Example 2. Let us take a mobile phone application that allows the reception of video calls. The application provider wants to ensure that video calls use the front camera of the phone, which motivates having two adaptations `Video (V)` and `FrontCamera (F)`. A **strong inclusion** relation $S(\mathtt{V}, \mathtt{F})$ is defined between the two adaptations (`V`–►`F`). Further, as receiving video calls is power consuming, the video call functionality should only be available if the battery is sufficiently high, which can be modeled through a `HighBattery (H)` adaptation. A **requirement** relation $R(\mathtt{V}, \mathtt{H})$ is defined between the two adaptations (`V` –◄`H`). The CoPN, $\mathcal{P} = \circ(S, R)$ obtained by the composition of the two dependency relations is shown in Fig. 16.

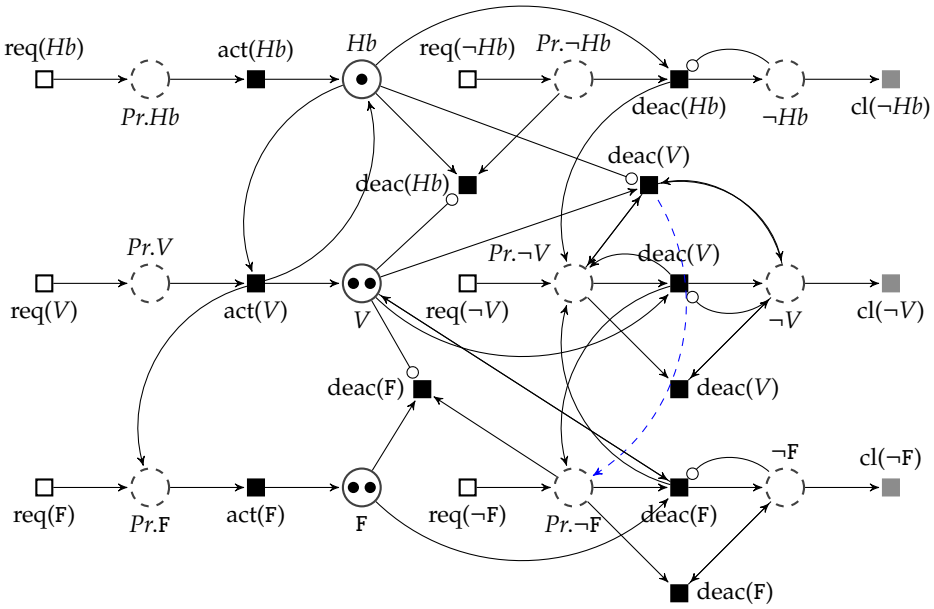


Fig. 16: Composition of a strong inclusion and a requirement dependency relation.

Based on the constraints specified by the set $\mathscr{C}_S$ and in particular the seventh clause in Definition 5.2, where every transition deactivating the source adaptation must deactivate the target adaptation, the dashed arc $(deac(\mathtt{V}), Pr.\neg\mathtt{F})$ is added to $\mathcal{P}$ in Fig. 16.[11] All other arcs and transitions come from the CoPNs for $S(\mathtt{V},\mathtt{F})$ and $R(\mathtt{V},\mathtt{H})$.

Let $\mathcal{P}$ be a CoPN defined by the adaptations C_i where $i \in I$ and I is an index set. Let C be an adaptation not in $\mathcal{P}$ such that there is a dependency relation $R_1(C, C_i)$ for $C_i \in \mathcal{P}$. In order to compose dependency relation R_1 into $\mathcal{P}$ it is sufficient to take any other relation R_2 such that $R_2(C_i, C_j)$ exists for some $C_j \in \mathcal{P}$. Let $\mathcal{P}'$ be the CoPN obtained by extending $\mathcal{P}$ with R_1 —that is, the result of the composition $\circ(R_1, R_2)$. Composing the relations R_1 and R_2 does not ensure that all constraints imposed by all dependency relations are satisfied by $\mathcal{P}'$. Using Definition 9, it can be ensured that $\mathcal{P}' \models \mathscr{C}_{R_2} \wedge \mathscr{C}_{R_1}$. However, to ensure that all of the pre-exiting constrains are still satisfied, we use the process explained in Section 3 for all such constrains and the ones introduced by relation R_1 in the new CoPN —that is, ensure that $\mathcal{P}' \models (\bigwedge_I \mathscr{C}_{R_i}) \wedge \mathscr{C}_{R_1}$ is satisfied.

The fact that adaptation activations and deactivations can only be requested through temporary places means that a adaptation only interacts with the adaptations immediately related to it. If dependency relations are transitive (e.g. weak inclusion) then the request for activation takes place for the adaptations

[11] Here the arc is dashed as a mean to easily identify it. This convention has no especial semantic meaning in the CoPN.

immediately related to the adaptation being activated, which then forward such requests to their immediate related adaptations and so on. For example, in the chain A–▷ B–▷ C, of weak inclusions, activation of adaptation A request activation of adaptation B, and activation of adaptation B request the activation of adaptation C. If an adaptation D is composed with C (C–▷ D), when A is activated the responsibility of activating D is delegated to C (as it is its immediate relation).

If temporary places would not exist, then the composition of an adaptation would affect the CoPN globally, forcing the addition of transitions and arcs to many adaptations that are not immediately related to the adaptation being composed. In the chain of weak inclusions, for example, arcs would have to be added between A and B, C, and D.

Theorem 2. *The ∘ operator is idempotent, associative and commutative.*

Proof. These properties follow trivially from the idempotence, associativity and commutativity properties of the ∧ operator.

In principle it is always possible to compose two CoPNs. However, composition of adaptations does not always yield a coherent CoPN with respect to its behavior. For example, composing two strong inclusion dependency relations (A–►B) and (B–►A), yields a CoPN with an infinite loop between the activation transition of adaptation B and the activation transition of adaptation A. The *identification* of such incoherent or erroneous CoPN behavior is part of our future work. A discussion on how these problems can be addressed is discussed in the following section.

7 Alternative Approaches

This section overviews related modeling approaches. A comparison is presented in terms of three criteria that are relevant in the case of dynamically adaptable systems: expressiveness, run-time capabilities, and analysis and verification tools. In the light of these criteria, the following sections present firstly the model that inspired our adaptation dependency relations; secondly, alternative state transition models; and thirdly, a number of related formalisms that have been used to define software systems. The final section presents a discussion, guided by the aforementioned criteria, which puts in perspective our choice of Petri nets.

7.1 Feature-Oriented Development and Feature Interaction

The dependency relations presented in this paper are inspired on those defined in feature diagrams [20]. Dependencies are normally used to declare interactions between different features or behavioral variation modules in software product lines [6]. Feature interaction [**?**] is concerned with the identification of inconsistencies that may rise from the activation and interaction of different feature modules, based on the defined dependencies between them. In the general

case, feature interaction is a complicated problem, and it becomes even harder when addressed for dynamic settings. Recent approaches address the dynamic feature interaction problem by the upfront definition of policy rules between features that may interact at run time [22,10]. Whenever a feature is included at run time in the run time [10]. Whenever a feature is included at run time in the program, the policies are verified. In case there is an interaction problem, the associated corrective process defined by the policy is applied. This approach has as main drawback that policies and resolution rules need to be predefined, and they therefore cannot cover all possible interactions of the system. Verification of policies and conflict resolution are costly operations, which are unsuitable for dynamically adapting systems.

7.2 State Diagrams

Automata [18] and statecharts [21] 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 and state charts are normally used to verify system properties, such as program termination. Among the properties provided by these diagrams, the most prominent is the available operations: automata can be easily merged, intersected and composed in a parallel manner. However, at a particular moment of execution, the system focuses only on one particular state, this means that the state needs to associate all possible actions in the system to that particular state. For the particular case of context-aware systems, actions such as context activations can be triggered non-deterministically based on the surrounding environment of execution. This would mean to have all possible activations incident to every state of the automata, making it cluttered and difficult to manage.

7.3 Process Algebra, Coalgebra and Modal Logics

Different mathematical formalisms have also been used to model and procure formal reasoning analysis about software systems. Process algebra [16] is used to model concurrent processes, providing high-level abstractions for operations between processes such as parallel composition, communication, replication, and synchronization. Modal logics [3] have been used to represent necessity and possibility conditions about system properties. Modal logics are mostly used to express temporal conditions, but they also can be used to express conditions like termination of programs in the case of propositional dynamic logic. Coalgebra, and in particular coalgebraic specification [19], has been used to express the dynamic behavior of systems. Typically, coalgebras specified state-based systems, where the state is considered as a black box and dynamic behavior is reasoned upon in terms of invariance and bisimilarity. Modal operators have also been introduced in coalgebraic specifications as invariants to reason about future states in safety and progress formulas.

These three formal methods on their own are used as an abstract model to prove consistency or decidability properties about the systems they model.

However, concrete models based on the formalisms can be defined for each of them. Examples of these concrete models are: abstract state machines [4], algebraic petri nets [11], alternating automata[28] or computational tree logic [7]. Regardless of the implementation model, verification and analysis of system properties is often done offline by means of model checking techniques.

7.4 Model Checking

Model checking is an analysis and verification technique that is transversal to all the approaches presented above (including Petri nets). All approaches can benefit from offline model checking to prove some of the system properties. Model checking techniques include, abstract interpretation, partial order reduction, or automated theorem proving. Here we discuss the SAT, which is commonly used to decide over system properties defined by a (set of) logic formula(s) [26]. Based on the semantic definitions of dependencies described in Section 5, formulas describing the activation and deactivation of adaptations could be generated. Each (de)activation would be verified by the SAT solver, and if it were satisfiable, then the adaptation could be (de) activated. In order for the verification to take place, it would be necessary to provide the state of the system (usually done with an automata), which could greatly increase the number of formulas needed.

7.5 Discussion

Now, we turn to the criteria defined at the beginning of this section to evaluate each of the approaches.

Feature models express the different dependency rules between features, and their interaction much in the sense it is done for context-oriented systems. However, these interactions are normally managed offline, which is not desirable to deal with dynamic adaptations at runtime, even then, a complete run of a SAT solver, or the verification of interaction resolution policies for each adaptation (de)activation can be very costly for the runtime execution of a system.

Petri nets, in some cases have an equivalent automata representation [5]. Representing systems with automata is particularly useful because of their decidability results, used to verify properties about the systems they model. However, runtime verification of automata can be cumbersome, because most of the analysis is intended to be done offline. Another contra not to use automata (or one of its similar models) as an execution model for context-aware systems, is that, in such model all possible transitions between states must be considered, which would make the automata too cluttered and difficult to manage. Additionally, coding the activation counter property, presented in Section 2 would need to be done by an extra extension of the automata model.

Formal methods, such as coalgebras or modal logics can also provide an effective formalization for context-aware systems. However, being theoretical models they cannot be use for the runtime representation of the system, or to

effectively verify adaptations (de)activation, that is, without undergoing complicated (model) checks, or additionally having to generated corresponding automata structures, making them ill suited.

To conclude this discussion we argue that the Petri net model presented in the paper successfully capture a direct view of state (active adaptations) and dynamic properties (adaptation (de)activations) of a system. Moreover, it can effectively be used as an execution model. As compared to other approaches, Petri nets seem as a natural and good fit for the kind of dynamic systems envisioned by COP.

8 Future Work

The purpose of CoPN is to provide a sound semantics for COP systems, and to ensure consistency of such systems. CoPN gives a sound concurrency semantics to COP systems. Furthermore, it is possible to prove that, using this semantics, context activations and deactivations maintain system consistency.

We state that the verification of adaptation activation and deactivation becomes more lightweight in CoPNs. This statement is motivated by the activation process, as adaptation change their state lively —that is, without going first through all adaptations checking if the changes are possible. However, a full benchmark study on the efficiency of our CoPN model with respect to the existing Subjective-C implementation is still future work.

As mentioned in Section 6.1, CoPN composition can yield incoherences or erroneous behavior. Such errors must be identified whenever CoPNs are composed. Petri nets can be used for this purpose. Standard Petri net analysis techniques allow to reason about a system's behavior [23]. Such properties could be used to identify interaction between contexts, the properties that could be used in the context of COP systems comprise: 1. *reachability*, which could be used to identify if is possible to have a particular configuration of active contexts (i.e., marking), 2. *liveness*, which could verify if a context can ever be activated or not 3. *persistency*, which could spot isolated contexts in the Petri net, and 4. *deadlocks*, which could identify contexts that due to a configuration can never be active again. These analysis can give upfront information about errors or redundancies in the system. the CoPN model contains inhibitor arcs and is (in principle) unbound which make these properties undecidable. However, restrictions to bound contexts and simplifications on the use of inhibitor arcs could be made in order to analyze such properties [27]. An study of which of the properties can be successfully verified for CoPN is part of our ongoing work.

A Petri net is said to be bounded (or k-bounded) if there is an integer k such that every place in the net can contain maximum k tokens. Bounded Petri nets are used to express that a resource is scarce. Although a bound could be imposed for some adaptations, in principle adaptations can be activated multiple times, making CoPNs unbound. Boundedness is an important property because it makes it possible to verify other properties such as reachability. Bounds could be defined in CoPN to allow the verification of such properties

within those bounds. The current implementation of CoPN already allows us to define bounds for adaptation activation. The analysis on how to properly choose such a bound is future work.

Reachability is used to verify if certain markings could ever occur in a Petri net given the initial marking. In the context of COP, such property could be used to identify if a particular configuration of active contexts (i.e. marking) is possible, given the current state of the system (i.e. initial marking). Reachability verification is usually undecidable for Petri nets with inhibitor arcs. However, there are special cases in which it is possible to verify reachability in such Petri nets [27]. The study of whether or not CoPN complies with these conditions, and the development of the reachability verification algorithm, is part of our ongoing work.

Another ongoing task is the verification of liveness (in its stronger version). Liveness means that no matter the marking of a Petri net, it is always possible to eventually fire all of its transitions. In the context of COP, this could be used to verify if context activations (transition firings) can ever take place. That is, if a context can ever be activated or deactivated given the initial system state.

9 Conclusions

Management of highly dynamic adaptations as proposed by Context-Oriented Programming (COP) has proven a challenging task. The composition of adaptations may lead to unexpected or contradictory behavior if not dealt with carefully. To avoid these problems, different modeling techniques have been proposed. These techniques share the commonality of encoding constraints between adaptations by means of dependency relations, thereby forming what we call dependency graphs. Although dependency graphs constitute a useful first step towards the specification of adaptation dynamics in COP systems, they are not entirely satisfactory, specially because they lack a precise semantics.

With the objective of providing a more precise and concrete formalism for the expression of dependencies between adaptations, we propose the context Petri nets execution model. We show how to map the adaptations and dependencies of a high-level dependency graph as places and transitions in a context Petri nets (CoPN). In doing so, the semantic constraints imposed by the dependencies in the original model become more precise. Furthermore, since a CoPN provide a concrete and live representation of context-aware systems, it is a suitable model represent evolution of adaptations at runtime. To allow the dynamic activation, deactivation, and composition of adaptations, the constraints imposed by their relations must be verified. In dependency graphs this computation can have a considerable computation cost, but in CoPN the verification of dependencies between adaptations is more lightweight, since it boils down to checking the input places of transitions.

For the advantages they bring in the modeling of adaptation dynamics, context Petri nets seem to be a convenient formalism both for modeling and run-time representation of adaptations and their constraints in COP systems.

References

1. Bause, F.: On the analysis of petri nets with static priorities. In: Acta Informatica. vol. 33, pp. 669 – 685 (1996)
2. Best, E., Koutny, M.: Petri net semantics of priority systems. Theoretical Computer Science 96, 175–215 (April 1992), http://dx.doi.org/10.1016/0304-3975(92)90184-H
3. Blalckburn, P., de Rijke, M., Venema, Y.: Modal Logic. Cambridge University Press (2001)
4. Borger, E., Stark, R.: Abstract State Machines. Springer-Verlag (2003)
5. Cassandras, C.G., Lafortune, S.: Introduction to Discrete Event Systems, pp. 225 – 273. Discrete event dynamic systems, Kluwer Academic Publishers (2007)
6. Cetina, C., Haugen, O., Zhang, X., Fleurey, F., Pelechano, V.: Strategies for variability transformation at run time. In: Proceedings of the International Software Product Line Conference. pp. 61–70. Carnegie Mellon University, Pittsburgh, PA, USA (2009)
7. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. 8, 244–263 (April 1986)
8. Costanza, P., D'Hondt, T.: Feature descriptions for context-oriented programming. In: Thiel, S., Pohl, K. (eds.) 12th International Software Product Line Conference, Second Volume (Workshops). pp. 9–14. Lero Int. Science Centre, University of Limerick, Ireland (2008)
9. Costanza, P., Hirschfeld, R.: Language constructs for context-oriented programming: an overview of ContextL. In: Proceedings of the Dynamic Languages Symposium. pp. 1–10. ACM Press (Oct 2005), collocated with OOPSLA'05
10. Desmet, B., Vallejos, J., Costanza, P., De Meuter, W., D'Hondt, T.: Context-oriented domain analysis. In: Modeling and Using Context. pp. 178–191. Lecture Notes in Computer Science, Springer-Verlag, Berlin, Heidelberg (2007)
11. Dimitrovici, C., Hummert, U., Petrucci, L.: Semantics, composition and net properties of algebraic high-level nets. In: Rozenberg, G. (ed.) Advances in Petri Nets 1991, Lecture Notes in Computer Science, vol. 524, pp. 93–117. Springer Berlin / Heidelberg (1991)
12. Eshuis, R., Dehnert, J.: Reactive petri nets for workflow modeling. In: Application and Theory of Petri Nets 2003. pp. 296–315. Springer (2003)
13. González, S.: Programming in Ambience: Gearing Up for Dynamic Adaptation to Context. Ph.D. thesis, Université catholique de Louvain (Oct 2008), http://hdl.handle.net/2078.1/19684, coll. EPL 211/2008. Promoted by Prof. Kim Mens
14. González, S., Cardozo, N., Mens, K., Cádiz, A., Libbrecht, J.C., Goffaux, J.: Subjective-C: Bringing context to mobile platform programming. In: Proceedings of the International Conference on Software Language Engineering. Lecture Notes in Computer Science, vol. 6563, pp. 246–265. Springer-Verlag (2011)
15. González, S., Mens, K., Heymans, P.: Highly dynamic behaviour adaptability through prototypes with subjective multimethods. In: Proceedings of the Dynamic Languages Symposium. pp. 77–88. ACM Press, New York, NY, USA (Oct 2007), collocated with OOPSLA'07
16. Hennessy, M.: Algebraic Theory of Processes. MIT Press, Cambridge, Mass. (1988)
17. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. Journal of Object Technology 7(3), 125–151 (March–April 2008)
18. Hopcroft, J.E., Ullman, J.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley (1979)

19. Jacobs, B.: Exercises in coalgebraic specification. In: Algebraic and Coalgebraic Methods in the Mathematics of Program Construction. pp. 237–280 (2000)
20. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (foda) feasibility study. Tech. Rep. CMU/SEI-90-TR-21, Carnegie-Mellon University Software Engineering Institute (Nov 1990)
21. Latella, D., Majzik, I., Massink, M.: Towards a formal operational semantics of uml statechart diagrams. In: Proceedings of the Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS). pp. 465–. Kluwer, B.V., Deventer, The Netherlands (1999)
22. Liu, Y., Meier, R.: Resource-aware contracts for addressing feature interaction in dynamic adaptive systems. Autonomic and Autonomous Systems, International Conference on 0, 346–350 (2009)
23. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE 77(4), 541 – 580 (April 1989)
24. Peterson, J.L.: Petri nets*. In: Computing Surveys. vol. 9, pp. 223 – 252. ACM (September 1977)
25. Pommereau, F.: Quickly prototyping petri nets tools with snakes. Petri net newsletter pp. 1–18 (October 2008)
26. Prasad, M.R., Biere, A., Gupta, A.: A survey of recent advances in sat based formal verification. Journal on Software Tools for Technology Transfer 7(2), 156 – 173 (2005)
27. Reinhardt, K.: Reachability in petri nets with inhibitor arcs. Electronic Notes in Theoretical Computer Science 223, 239–264 (2008)
28. Streett, R.S., Emerson, E.A.: An automata theoretic decision procedure for the propositional mu-calculus. Information and Computation 81(3), 249 – 264 (1989)