Vrije Universiteit Brussel

Faculty of Science and Bio-Engineering Sciences
Department of Computer Science
Software Languages Lab

Reactive Method Dispatch for Context-Oriented Programming

A dissertation submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy in Sciences (Computer Science)

Engineer Bainomugisha

Promoter: Prof. Dr. Wolfgang De Meuter

12 December 2012

Print: Silhouet, Maldegem

2012 Uitgeverij VUBPRESS Brussels University Press
VUBPRESS is an imprint of ASP nv (Academic and Scientific Publishers nv)
Ravensteingalerij 28
B-1000 Brussels
Tel. +32 (0)2 289 26 50
Fax +32 (0)2 289 26 59
E-mail: info@vubpress.be
www.vubpress.be

ISBN 978 90 5718 244 0
NUR 989
Legal deposit D/2012/11.161/177

Abstract

We are on the verge of a post-PC revolution where computing power is shifting from laptops and desktops to smartphones, tablets and everyday objects such as eyeglasses and coffee machines. Those computing objects come equipped with a myriad of context sensors such as an accelerometer, a proximity sensor, a gyroscope, a GPS receiver and NFC technology. Context sensors enable the development of context-aware applications that continuously adapt their behaviour to match the context of use. In this dissertation, we refer to applications that are *always* prepared to *promptly adapt* their behaviour in reaction to context changes as *reactive context-aware applications*.

Our research hypothesis is that developing *reactive* context-aware applications remains notoriously difficult because of the lack of suitable programming language abstractions coupled with the unpredictable nature of context changes. More concretely, current programming languages fall short of providing the appropriate support for developing context-aware applications that need to react promptly to a sudden context change – especially if such a context change occurs in the middle of an ongoing procedure execution. Such a context change may require an ongoing procedure execution to be *promptly interrupted* in order to prevent its execution from happening in a wrong context.

The vision of this dissertation is to investigate novel programming language abstractions to facilitate the development of *reactive* context-aware applications. We propose a new programming language model called *interruptible context-dependent executions*, where a procedure execution is always constrained to happen only under a developer specified context condition. In this model, the execution of a context-dependent procedure can be seamlessly interrupted or resumed depending on whether the specified context condition is satisfied or not. The language runtime ensures that the execution state of a context-dependent procedure is automatically preserved between interruptions. The model features a new dispatching mechanism called *reactive dispatching* that continually takes into account new context changes to select the applicable procedures and to suspend or interrupt ongoing procedure executions.

We present a proof-of-concept programming language called *Flute* that incorporates the interruptible context-dependent executions model. Flute has been implemented as a meta-interpreter on top of *iScheme* – a technological research artefact that we developed to enable the experimentation with novel language abstractions and features to ease the development of *reactive* context-aware applications. iScheme runs on the iOS mobile platform and as such it fosters experiments on real mobile devices.

Samenvatting (Abstract in Dutch)

We bevinden ons aan het begin van een ware post-PC revolutie waarin rekenkracht verschuift van computers naar tablets, naar mobiele telefoons en zelfs naar alledaagse objecten zoals brillen en koffiemachines. Deze objecten zijn voorzien van verschillende sensoren voor context zoals versnellingsmeters, nabijheidssensoren, gyroscopen, ontvangers voor GPS-signalen en NFC-technologie. Deze sensoren maken het ontwikkelen van context-bewuste applicaties mogelijk die hun gedrag voortdurend afstemmen op de context waarin ze gebruikt worden. In dit proefschrift verwijzen we naar applicaties die *op elk moment* ten gevolge van contextveranderingen hun gedrag *prompt* kunnen aanpassen als *reactieve context-bewuste applicaties*.

Onze onderzoekshypothese is dat het ontwikkelen van *reactieve* context-bewuste applicaties notoir moeilijk blijft door het gebrek aan geschikte programmeertaalabstracties en de onvoorspelbare aard van contextveranderingen. Bestaande programmeertalen bieden geen geschikte ondersteuning voor het ontwikkelen van context-bewuste applicaties die prompt moeten reageren op een plotse contextverandering –vooral wanneer deze zich voordoet tijdens de uitvoering van een procedure. Bij zulk een contextverandering kan het nodig zijn een lopende uitvoering te staken teneinde te voorkomen dat een procedure uitgevoerd wordt in de verkeerde context.

De visie van dit proefschrift is het ontwikkelen van *reactieve* context-bewuste applicaties te ondersteunen via nieuwe programmeertaalabstracties. Hiertoe stellen we een nieuw programmeertaalmodel voor genaamd *onderbreekbare context-afhankelijke uitvoeringen* waarin een door de ontwikkelaar opgegeven contextvoorwaarde de uitvoering van een procedure beschermt. Hierbij wordt de uitvoering van een context-afhankelijke procedure onderbroken wanneer de contextvoorwaarde niet langer voldaan is en weer naadloos hervat van zodra dit terug het geval is. De runtime van de programmeertaal zorgt ervoor dat de uitvoeringsstaat van een context-afhankelijke uitvoering automatisch bewaard blijft tussen de onderbreking en de hervatting. Kenmerkend voor dit programmeermodel is een nieuw dispatching mechanisme genaamd *reactive dispatching* dat contextveranderingen in acht neemt bij het selecteren van nieuwe uit te voeren procedures en bij het onderbreken of hervatten van lopende procedures.

Als bewijs van concept stellen we de programmeertaal *Flute* voor die het programmeermodel van *onderbreekbare context-afhankelijke uitvoeringen* incorporeert. Flute is gerealizeerd als een meta-interpreter in *iScheme* – een onderzoeksartefact dat we ontwikkeld hebben om het experimenteren met innovatieve programmeertaalabstracties voor *reactieve context-bewuste applicaties* te vergemakkelijken. iScheme bevordert experimenten op mobiele

apparaten via diens ondersteuning voor het iOS platform.

Acknowledgements

This dissertation would not be where it is today without tremendous support, advice, and encouragement that I received from a number of people. It is quite difficult to find the right words to thank everyone the way I want.

I would like to express my sincere thanks to my promoter Prof. Wolfgang De Meuter for giving me the opportunity to do my Ph.D. research at the Software Languages Lab – without him my Ph.D. research would not have started. His advice and support have been invaluable and I cannot begin to enumerate what I owe him. I have learnt a great deal from him and his passion for innovative programming languages research has been a constant source of inspiration and motivation for my Ph.D. research. I am also greatly indebted to my advisor, Dr. Coen De Roover for his input in my work and his useful advice during the past couple of years.

I would like to thank the members of my thesis committee, Prof. Robert Hirschfeld, Prof. Kim Mens, Prof. Sven Casteleyn, Prof. Bernard Manderick, and Prof. Theo D'Hondt for the discussion, questions and suggestions for improving this thesis. Besides being on my Ph.D. committee, Prof. Theo D'Hondt also introduced me to Scheme during my master studies. There is no doubt that his courses greatly shaped my view on programming languages. In addition, Prof. Theo D'Hondt's software artefacts (Skem, SLIP) have been a major foundation for my work.

I thank Prof. Wolfgang De Meuter, Dr. Coen De Roover, Dr. Jorge Vallejos, and Dr. Andoni Lombide Carreton for carefully reading the draft of this thesis and providing me with very helpful feedback and suggestions for improving this thesis. This thesis has significantly benefited from their input. I would also like to thank all the four of you for the brainstorming sessions and the time we spent together discussing the language ideas for *interruptions*, *modes* and *modals*. In addition, I thank Dr. Elisa Gonzalez Boix and Nicolas Cardozo for the help in preparing for my private defence. Dr. Coen De Roover further helped me to translate my abstract to Dutch.

I thank all the past and current colleagues of the Software Languages Lab for many joyful days and fruitful discussions. It has always been a pleasure to work in this group of extremely intelligent and friendly people. I thank Dr. Pascal Costanza for introducing me to context-oriented programming and also guiding me in the first years of my Ph.D. research. Dr. Jorge Vallejos for being a colleague, a friend, office mate and for the insightful discussions. Prof. Tom Van Cutsem for his contribution in securing the SAFE-IS project

This Ph.D. research was funded by the SAFE-IS project in the context of the Research Foundation - Flanders (FWO).

that has been funding my research. Prof. Viviane Jonckers for the opportunity to be an assistant of the Software Architecture course which provided teaching experience. Stefan Marr, Dr. Veronica Uquillas Gomez, and Dr. Andy Kellens for the collaboration on the Software Architecture course. I also thank the members of the Context-oriented programming and Ambient research group for the collaborations and knowledge-sharing reading groups. In particular, Dr. Andoni Lombide Carreton, Prof. Tom Van Cutsem, and Stijn Mostinckx on the reactive programming publication and Dr. Elisa Gonzalez Boix on the iScheme work.

I thank Koosha Paridel and Prof. Yolande Berbers for the collaboration in the context of the SAFE-IS project and the resulting joint publications. I would also like to thank all my master thesis and 3rd bachelor students, who participated in shaping some of my research artefacts. In particular: Nick De Cooman for building the first large case study using iScheme. Christian Ivàn Hernàndez, Pieter Mensalt, David Meert, and Simon De Schutter whose theses greatly shaped iScheme and Flute.

Being part of a sports team is important for surviving Ph.D. research. I thank the members of the badminton and volleyball teams for the sports evenings and beach-volley weekends. The secretaries (Lydie, Simonne, and Brigitte), I thank you for helping with the administration and accounts. I also want to thank all the friends and colleagues that I have met in Belgium who have in one way or another contributed to the success of my stay in Belgium: Emmanuel, Marie, John Mark, Sonia, Steven, Sylvie, Ilse ... I am also very grateful to my friends and colleagues in Uganda who have encouraged me or provided me with practical support despite the geographical distance: Rogers, Pelga, Maali, Fiona, Rossette, Ronnie, Bigzo, Simon, Steve, Ivan, Evans, Brian, Agaptus, Moses ...

To my family, I am immensely grateful. First of all in the memory of my parents and sister. Special thanks to my sisters (Eva and Yuria) my brothers (Norbert and Jonan), my grandparents (every time I left Uganda to come to Belgium they wondered whether I will be able to see them again), my uncles (of special mention Asiimwe) and aunties for the continued support and encouragement throughout my Ph.D. research.

Needless to say, any remaining errors, oversights and any other deficiencies are mine and mine alone.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

1.1 Introduction

This dissertation seeks to develop novel programming language abstractions for an emerging class of mobile applications that we call *reactive context-aware applications*. By *reactive* context-aware applications we mean applications that sense their environment and *promptly adapt* their behaviour in order to match their physical context of use. A key enabler for *reactive* context-aware applications is the rapid advancements in computer hardware technology where computing devices come equipped with a myriad of context sensors such as a GPS receiver, an accelerometer, a proximity sensor, a gyroscope, and Near Field Communication (NFC) technology [LML+10]. The explosion of sensor-equipped devices has led us to believe that the future of mobile applications lies in *reactive* context awareness. The presence of sensors on mobile devices is fundamentally changing the way users interact with software applications. Instead of *user-initiated interactions* (e.g.,

a pinch gesture on a touch screen to launch an application), we are moving towards *context-driven interactions* where context sensors become the driver for deciding which application or behaviour to run for the current task at hand (e.g., a device detecting that a user is nearby a bus stop and automatically launching the application to show the bus schedule for that stop). It is not hard to see the potential of such context-driven interactions in the near future. Envision, for instance, a mobile operating system that is enhanced with context awareness to perpetually switch between applications and application behaviour to match the changing user's context without explicit user intervention. In Chapter 2, we will present a visionary scenario that substantiates our claims.

Recently, programming languages for context-aware applications have emerged [HCN08, Val11, Gon08, SGP12a]. However, our survey of the state of the art (cf. Chapter 3) reveals that current programming languages fall short of providing suitable support to ease the development of *reactive* context-aware applications. Our research hypothesis is that the lack of suitable programming language abstractions coupled with the unpredictable nature of context changes renders the task of developing *reactive* context-aware applications notoriously difficult. In short, current programming languages fail to provide adequate support for developing context-aware applications that need to *react promptly* to *a sudden context change* – especially if such a context change occurs in the middle of an ongoing procedure execution. Currently, developers have little choice but to resort to *explicit management of the execution state* (saving and restoring application execution state between context changes) and *explicit context checks* (to ensure that the procedure execution is always constrained to run only in the correct context). Such manual approaches are error-prone and may lead to incorrect application behaviour. The research presented in this dissertation aims at liberating developers from the bane of developing *reactive* context-aware applications.

The vision of this dissertation is to investigate novel programming language abstractions to facilitate the development of *reactive* context-aware applications. We propose a new programming language model that we call *interruptible context-dependent executions*. The primary concept of *interruptible context-dependent executions* is that a procedure execution should *always be constrained* to happen only under particular context conditions. In this model, the execution of a context-dependent procedure is seamlessly interrupted or resumed depending on whether the specified context condition is satisfied or not. In addition, the execution state of a context-dependent procedure is automatically preserved between interruptions. We present a new programming language called *Flute* that epitomises the interruptible context-dependent executions model.

To ground our vision, we will first sketch the research context of our work by positioning this dissertation in existing research domains. We will then present the problem statement that details the issues that we need to tackle in this research. After stating the problem statement, we will formulate the research questions that need to be addressed and then present the main contributions of our research. We conclude this introductory chapter with a roadmap to guide the reader through the entire dissertation.

1.2 Research Context

We will briefly introduce four research domains that form the background of the research that we present in this dissertation. These include *ubiquitous computing, ambient intelligence, context-aware computing,* and *programming language design.*

Ubiquitous Computing. Nearly two decades ago Mark Weiser proposed the vision of *Ubiquitous Computing* where computers become invisible to the user and are integrated into tiny and cheap everyday objects [Wei93, Wei95]. At the time, Weiser's vision seemed like science fiction. However, this vision is becoming a reality thanks to the rapid advances in the hardware technology. Today, computing devices are available in various sizes and shapes including household objects (a phenomenon Jessie Dedecker described as "hardware miniaturisation" in his Ph.D. dissertation [Ded06]). As we write this dissertation, researchers are working towards a smaller and cheaper GPS receiver dubbed the "GPS Dot" that gives a centimetre level precision [Hum12]. Such device miniaturisation and sensor precision are key enablers for developing *reactive* context-aware applications, which is the theme of this dissertation. Indeed, Weiser noted that if ubiquitous computers know their environment, they can automatically adapt their behaviour in significant ways [Wei95].

Ambient Intelligence. *Ambient Intelligence* (AmI) is a vision of a future in which environments are sensitive and responsive to the presence of people inhabiting them [Far11]. The term AmI was coined by the European Community's Information Society Technology Group (ISTAG) [Gro03]. It builds upon the vision of ubiquitous computing. In this vision environments are interconnected, adaptable, dynamic, embedded, and intelligent and are capable of anticipating the needs and behaviour of their inhabitants. Most of the AmI scenarios are targeted towards Smart Homes. A Smart Home environment is equipped with sensors and actuators for controlling appliances and

equipment. Examples of Smart Home services include supporting independent living for the elderly and the impaired (e.g., task execution monitoring and supervision, monitoring the use of medication and sleep). Sadri [Far11] performed a comprehensive survey of the applications of AmI and observes that there is a need for context-sensitivity in most of the AmI systems. In the elderly care systems, for instance, the action to perform is based on the context situation (e.g., adjusting heaters based on the temperature of the surroundings or suggesting a new schedule of activities based on the user's need).

Context-aware Computing. Context-aware computing is an enabling technology for both ubiquitous computing and ambient intelligence that focuses on systems that sense their surrounding context and adapt their behaviour accordingly [SAW94]. A prevalent example of contextual information is location. However, context is more than location. It may include other information such as nearby people and objects, time of the day, user preferences, communication bandwidth, noise level or even social aspects such as the age group of the people around you. Developing context-aware applications poses a number of challenges that range from context acquisition from the environment, reasoning about raw contextual data, and adapting the application behaviour accordingly. In this dissertation, we focus on investigating enabling programming language techniques for *prompt adaptation* of context-aware applications in reaction to context changes.

Programming Language Design. At the software engineering level, research on context awareness has mainly been undertaken along two paths, namely, middleware approaches [DAS01, CEM03, BDR07] and programming language approaches [HCN08, Val11, Gon08, SGP12a]. In this dissertation, we pursue a programming language-based approach by investigating the design of novel linguistic abstractions that ease the development of *reactive* context-aware applications. The research presented in this dissertation was conducted at the Software Languages Lab of the Vrije Universiteit Brussel, Brussels, Belgium. The lab has a strong culture of programming language design and over the past years a number of experimental programming languages targeted for the domains of ubiquitous computing, ambient intelligence, and context-aware computing have been developed. These include ChitChat [De 04], AmbientTalk [Van08], AmbientTalk/R [Lom11], Lambic [Val11], and ContextL [HCN08]. Therefore, it is evident that the path of the programming language approach which has been taken in this dissertation has been influenced by the language design expertise that is lo-

cally available. Nevertheless, we are not arguing against the middleware approach. Instead, we believe that the two approaches complement each other. For instance, the programming language abstractions that we develop in this dissertation can ease the development middleware for *reactive* context-aware applications.

1.3 Problem Statement

Even though *reactive* context-aware applications provide end-users with enhanced experiences and richer application behaviour, the software technology for developing these applications is still underdeveloped. We argue that the lack of suitable programming language abstractions coupled with the **unpredictable nature of context changes** renders the task of developing *reactive* context-aware applications notoriously difficult. More concretely, current programming languages fall short of providing the appropriate support for developing context-aware applications that need to **react promptly to a sudden context change** – especially if such a context change occurs in the middle of an ongoing procedure execution. Current programming languages are based on the assumption that context changes that occur during a procedure execution will not immediately affect the application behaviour. Those assumptions are valid for traditional mobile and desktop applications, but not for *reactive* context-aware applications since context changes can occur at any moment in an unpredictable fashion.

Because context changes can occur at any moment during a procedure execution, it is possible that a procedure execution that started in a correct context may end up running in the wrong context. Allowing a procedure execution to continue executing in a wrong context may result in incorrect application behaviour (e.g., presenting to the user the application behaviour that does not match the current context). Currently, there is no suitable language support that enables developers to **constrain an entire procedure execution to a particular context**. In order to constrain an *entire* procedure execution to a correct context, developers of context-aware applications have to perform regular explicit context checks in the procedure body. A disadvantage with such an approach is that context checks need to be inserted throughout the procedure body. This leads to negative effects on program comprehension and maintainability (e.g., introducing a new context source implies modifying multiple existing context checks). In addition, the developer has to manually express concerns of the decisions to perform when such context checks are *not* satisfied. For instance, in order to be able to restore the execution later on, the developer has to devise means to capture

and restore the procedure's execution state. Finally, because program executions may be interrupted in reaction to context changes, the developer needs to ensure that the execution environment remains in a consistent state (e.g., controlling the visibility of state changes among procedure executions). Such concerns are not trivial and it is almost impossible to express them manually due to the unpredictability of context changes.

In Chapter 2, we further discuss the above problems with concrete scenarios. We will assess the inadequacy of the current programming languages for developing context-aware applications in Chapter 3.

1.4 Research Questions

To concretise the above problems, we put forward six research questions:

RQ. 1 *How to constrain an entire procedure execution to only occur under a particular context condition?*

RQ. 2 *What should happen when a context change occurs in the middle of an ongoing procedure execution? Should the execution be interrupted and possibly be resumed later on?*

RQ. 3 *How to ensure that a procedure execution resumes in a consistent environment even when the execution was interrupted before its completion?*

RQ. 4 *How can context be represented in the underlying programming language such that it can be manipulated, reacted upon, and combined with other programs?*

RQ. 5 *How can context-dependent behaviours be expressed such that new unanticipated behavioural variations can be added as required without requiring modification of existing behaviour definitions?*

RQ. 6 *How can a dispatching process for selecting context-dependent behaviours to execute be scheduled such that it takes into account the current context as well as future context changes?*

In Section 2.6 we refine the above research questions into requirements that should be satisfied by programming language features designed for *reactive* context-aware applications. Those requirements serve as the basis of the programming language model, *interruptible context-dependent executions*, that we propose in Chapter 4.

1.5 Contributions

The primary goal of our research is to design and develop a programming language to facilitate the development of *reactive* context-aware applications. The programming language should tackle the problems that we highlighted in Section 1.3 and should address the research questions that we put forward in Section 1.4. This dissertation achieves that goal through four contributions that we describe below:

The interruptible context-dependent executions model. The first contribution of our research is the interruptible context-dependent executions model [BVDR+12] that defines the properties and boundaries of a programming language for *reactive* context-aware applications. The interruptible context-dependent executions model is defined by the characteristics of *reactive* context-aware applications, which include *context-constrained executions*, *prompt adaptability* and *sudden interruptibility*. These characteristics are described in Section 2.3 and further refined into a set of programming language requirements for *reactive* context-aware applications that we describe in Section 2.6. The programming language requirements for *reactive* context-aware applications are: *R*.1 *Chained context reactions*, *R*.2 *Context-dependent interruptions*, *R*.3 *Context-dependent resumptions*, *R*.4 *Contextual dispatch*, *R*.5 *Reactive dispatch*, *and R*.6 *Reactive scope management*. To the best of our knowledge there is no existing approach that satisfies all the requirements. The interruptible context-dependent executions model satisfies these requirements by ensuring that a procedure execution is always constrained to only happen under a particular context condition. In this model, the execution of a context-dependent procedure is seamlessly interrupted or resumed depending on whether the specified context condition is satisfied or not. In addition, the execution state of a context-dependent procedure is automatically preserved between interruptions. We present the interruptible context-dependent executions model in **Chapter 4**.

A mobile language experimentation laboratory. The second contribution of our research is a language laboratory that we developed to facilitate the experimentation with novel language constructs and features for *reactive* context-aware applications. We engineered a language laboratory called *iScheme* [BVB+12], which blends the rich programming properties of the Scheme language [Ken96] and a state-of-the-art mobile device that is equipped with context sensors to enable realistic experiments. For our experiments, we chose Apple's iOS devices that include the iPhone smartphone

and the iPad tablet. iScheme supports a language symbiosis between Scheme and Objective-C [Koc09] (the mainstream language used for iOS development) by way of a reflective application-programming interface (API), which facilitates access to the iOS APIs in an event-driven style. Additionally, iScheme provides event-driven distribution constructs specially tailored for distributed mobile computing environments. In particular, the distribution constructs are based on the *ambient-oriented programming model* [Ded06] and have built-in support for peer-to-peer service discovery, asynchronous remote messaging, and failure handling. We present iScheme in **Chapter 5**.

A programming language for *reactive* context-aware applications. The third contribution of our research is a proof-of-concept programming language that incorporates the interruptible context-dependent executions model. We present a new programming language called *Flute* [BVDR+12] that adheres to the interruptible context-dependent executions model. The Flute language is implemented as a meta-interpreter on top of iScheme. The Flute language enables the developer to specify under what context conditions a procedure should be executed (by means of a single context predicate) and the language runtime ensures that the context predicate is respected throughout the procedure execution. In addition, Flute enables developers to specify what should happen when the context predicate is no longer satisfied (e.g., suspend or abort the execution) and what should happen when the context predicate later becomes satisfied again (e.g., resume or restart the execution). Developers can scope state changes made during the procedure execution by means of state management strategies provided by Flute. Flute features a new dispatching mechanism called *reactive dispatching* that continually takes into account new context changes to select applicable procedures to execute. The Flute language is presented in **Chapter 6**, its executable semantics is described in **Chapter 7**, and is validated in **Chapter 8**.

A visionary mobile application model. The fourth but not least contribution of our research is a visionary mobile application model where applications are automatically launched depending on contextual information without requiring the user to explicitly tap an icon to launch the application. In this application model, mobile platforms are enhanced with context awareness to perpetually switch between applications depending on the ever-changing user's context. Moreover, when there a switch between applications, the current application's state is automatically saved and the application is able to resume running from where it left off when the user goes back to the previous context. The vision of this mobile application model is presented

in **Chapter 2**. In **Chapter 8**, we present a prototype implementation of a mobile platform called the *iFlute platform* that epitomises that mobile application model.

1.6 Supporting Publications

The research presented in this dissertation is supported by peer-reviewed publications of the author [BLV+12, BVDR+12, BVB+12, BCC+11, BPV+12, TCW+12, BVT+09, BM12, BDMD09, VGBB+08, PBV+10]. Below we list the most relevant publications.

- **Journal:** [BLV+12] Engineer Bainomugisha, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. A survey on reactive programming. To appear in *ACM Computing Surveys*, published by: ACM. (**Chapters 3, 6** and **7**)

- **Conference:** [BVDR+12] 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 *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software Proceedings*, SPLASH/Onward! '12, Tucson, Arizona, USA, published by: ACM, 2012. (**Chapters 4** and **6**)

- **Journal:** [BVB+12] Engineer Bainomugisha, Jorge Vallejos, Elisa Gonzalez Boix, Pascal Costanza, Theo D'Hondt, and Wolfgang De Meuter. Bringing Scheme programming to the iPhone – Experience. *Software, Practice Experience.*, 42(3):331-356, published by: John Wiley & Sons, Inc., 2012. (**Chapters 6** and **8**)

- **Book chapter:** [BCC+11] Engineer Bainomugisha, Alfredo Cádiz, Pascal Costanza, Wolfgang De Meuter, Sebastián González, Kim Mens, Jorge Vallejos, and Tom Van Cutsem. Language engineering for mobile software. In Paulo Alencar Donald Cowan, editor, *Handbook of Research on Mobile Software Engineering: Design, Implementation and Emergent Applications*, published by: IGI Global, 2011. (**Chapter 3**)

- **Conference:** [BPV+12] Engineer Bainomugisha, Paridel Koosha, Jorge Vallejos, Yolande Berbers, Wolfgang De Meuter. Flexub: Dynamic Subscriptions for Publish/Subscribe Systems in MANETs. In

Proceedings of 12th IFIP International Conference on Distributed Applications and Interoperable Systems, Lecture Notes in Computer Science, Stockholm, Sweden, published by: Springer, 2012. (**Chapters 2** and **5**)

- **Conference:** [TCW+12] Eddy Truyen, Nicolas Cardozo, Stefan Walraven, Jorge Vallejos, Engineer Bainomugisha, Sebastian Günther, Theo D'hondt, Wouter Joosen. Context-oriented Programming for Customizable SaaS Applications. In *Proceedings of 27th ACM Symposium on Applied Computing*, Trento, Italy, published by: ACM, 2012. (**Chapter 3**)

- **Conference:** [BVT+09] Engineer Bainomugisha, Jorge Vallejos, Eric Tanter, Elisa Gonzalez Boix, Pascal Costanza, Wolfgang De Meuter, Theo D'hondt. Resilient Actors: A Runtime Partitioning Model for Pervasive Computing Services, In *Proceedings of International Conference on Pervasive Services (ICPS'09)*, London, UK, published by: ACM, 2009. (**Chapters 1** and **9**)

1.7 Roadmap

In Section 1.5, we outlined the contributions that this dissertation makes to the existing research on programming language support for mobile applications. Naturally, the structure of the remainder of this dissertation is aligned with the above contributions.

Chapter 2: The Vision of Reactive Context-aware Applications. This chapter grounds the vision of this dissertation by sketching a visionary scenario called *BainomuAppies in Kampala*. The scenario is about a digital platform that runs on buses and minibuses (a.k.a matatus) in Kampala. The onboard digital platform is enhanced with context awareness to show useful information to passengers by running a suite of interesting applications depending on contextual parameters such as the geolocation of the bus, the proximity of other buses, the proximity of certain stops and the identity of the customers that happen to be onboard the bus at a certain moment in time. From this scenario, we derive the key characteristics that are exhibited by *reactive* context-aware applications. Based upon these characteristics we put forward requirements that a programming language designed for *reactive* context-aware applications should adhere to.

Chapter 3: Programming Technologies for Context-aware Applications: State of the Art. In this chapter, we survey the state of the art of programming languages and techniques for context-aware applications. We sort the existing approaches into four categories: context-oriented programming languages, first-class continuations, coroutines, and threads, functional reactive programming languages, and other programming language facilities, which include guards, assertions and invariants. We evaluate each approach against the programming language requirements that we put forward in Chapter 2. The evaluation reveals that none of the existing approaches satisfies all the requirements.

Chapter 4: Interruptible Context-dependent Executions. This chapter presents our novel programming language model called *interruptible context-dependent executions* (ICoDE) that aims to satisfy requirements that we put forward in Chapter 2. We discuss the main properties of the ICoDE model, that is, *predicated procedures, reactive dispatching, interruptible and resumable executions, scoped state changes,* and *representation of context as reactive values*. For each property, we discuss design considerations that need to be taken into account in supporting it in a concrete programming language. We conclude this chapter by providing a distillation of the ICoDE properties and their design considerations into a set of properties that a programming language for *reactive* context-aware applications should satisfy.

Chapter 5: iScheme: A Laboratory for Mobile Programming Languages. In this Chapter 5, we present iScheme, which is an experimentation platform that we developed to facilitate experiments with novel language constructs and features for *reactive* context-aware applications. Central to iScheme, is the language symbiosis between Scheme and Objective-C, which facilitates the development of dynamic mobile applications that exploit context sensors available on the iOS devices. iScheme lays a foundation for the language constructs and features that we present in Chapter 6.

Chapter 6: The Flute Language: A Developer's Perspective. This chapter builds upon Chapters 4 and 5. In this chapter, we present the Flute, which is the first instantiation of the ICoDE model. Flute has been implemented as a meta-interpreter in iScheme. It provides language constructs for realising the properties of the ICoDE model. Throughout this chapter we demonstrate the language constructs using a context-aware calendar application. We conclude this chapter by

mapping the language constructs provided by Flute onto the programming language requirements that we put forward in Chapter 2.

Chapter 7: An Executable Semantics for Flute. This chapter describes an executable semantics for the Flute programming language through a meta-interpreter in iScheme. The Flute interpreter is implemented in a continuation-passing style [FW08], which explicitly passes along a *continuation* parameter. The continuation parameter makes the control flow explicit, which facilitates the capturing and saving the execution state of an expression at any step of the evaluation. The evaluation of the procedure body is broken down into sequences of expressions. At each evaluation step, the context predicate is re-evaluated to determine whether to proceed with the evaluation or not.

Chapter 8: Building Reactive Context-aware Applications using Flute. In this chapter, we validate the language constructs of Flute by implementing a mobile platform called the *iFlute platform*. The iFlute platform is enhanced with context awareness such that applications deployed on it are automatically launched depending on the current context of use. As part of the experiments we developed and deployed example applications, namely, a context-aware calendar application and a context-aware printer assistant, and a context-aware task guide.

Chapter 9: Conclusions and Future Work. In this chapter, we give a recap of our research contributions and a mapping of the programming language requirements for *reactive* context-aware applications, onto the Flute language. We also discuss the limitations of our approach and the possible future avenues of our research.

Chapter 2

The Vision of Reactive Context-aware Applications

Contents

2.1 Introduction

We are on the verge of a post-PC revolution where computing power is shifting from laptops and desktops to smartphones, tablets and everyday objects such as eyeglasses, and coffee machines. The increasing availability of sensors like accelerometers, proximity sensors, gyroscopes, GPS receivers and Near Field Communication (NFC) technology on those computing objects is fundamentally changing the way users interact with mobile software applications. We are moving away from *user-initiated interactions* to *context-driven interactions*, that is, context sensors become the driver for deciding which application or behaviour to run for the current task at hand without explicit user intervention. We are moving away from *formal interactions* to

casual interactions [Epp11], that is, user interactions with applications are characterised by frequent "in-between moments" such as the user temporarily playing a favourite video in the middle of an ongoing task. In this dissertation, we refer to applications that are context-driven and are *always* prepared for interruptions due to *sudden* context changes as *reactive context-aware applications*.

From the hardware perspective, technologies that enable the development of *reactive* context-aware applications are already available on most of today's mobile computing devices. However, software technologies are still lagging behind. For instance, the *app* model, which is embraced by most of today's mobile operating systems, still requires users to explicitly tap an icon in order to launch an application or select a certain behaviour of the application to run for the task at hand. We envision that in the near future a single device will run hundreds of applications and each of those applications will provide many variants of behaviours, which renders it extremely difficult for the user to explicitly select which application or behaviour to run for the task at hand. What is missing is a mobile application platform that is enhanced with context awareness in order to *perpetually switch* between applications or application behaviours to match the ever-changing user's context. Naturally, when there is a switch between applications or behaviours (e.g., due to an in-between moment), users expect applications to automatically save the current execution state and be able to resume running from where they left off at a later point in time. Obviously, we cannot expect the user to explicitly tap a save button (as in PC applications) from time to time in order to save the application's state. Applications should remember their execution state when there is a context change so as to be able to continue seamlessly when the user goes back to the previous context.

The lack of appropriate software technologies coupled with the unpredictable nature of context changes renders the task of developing *reactive* context-aware applications notoriously difficult. Even the context-oriented programming (COP) paradigm [HCN08] that was recently proposed for developing context-aware applications is still in its infancy and falls short of providing appropriate support for developing *reactive* context-aware applications described above (we review the state of the art in Chapter 3). As a consequence, developers have little choice but to resort to *explicit management of the execution state* (saving and restoring application execution state between context changes) and *explicit context checks* (to ensure that the procedure execution is always constrained to run only in the correct context). However, the unpredictable nature of context changes renders it almost impossible for the developer to know beforehand at which points in the program to implement the above concerns.

We want to liberate developers from the bane of developing *reactive* context-aware applications. It is therefore the vision of this dissertation to conceive a novel software technology that facilitates the development of *reactive* context-aware applications. To ground our vision, we will first introduce a scenario from which we derive the requirements for the desired software technology.

2.2 A Visionary Scenario: BainomuAppies in Kampala

BainomugiStar is a Kampala-based[1] public transport company. They operate several buses and minibuses (a.k.a matatus) on a number of lines all over the Kampala region. The attractive part of the BainomugiStar bus network is that their buses and minibuses are equipped with an onboard digital platform that shows useful information to the passengers by running a suite of interesting *apps* ("BainomuAppies"). The current app running as well as the kind of the information shown depends on contextual parameters such as the geolocation of the bus, the proximity of other BainomugiStar buses, the proximity of certain stops and the identity of the passengers that happen to be onboard the bus at a certain moment in time.

The BainomugiStar Company has a network that consists of two types of stops all over the Kampala region:

Bronze Stops are stops that merely consist of a pole showing the BainomugiStar-poster, e.g., somewhere on the sidewalk (cf. Figure 2.1). Technically speaking, the BainomugiStar-poster contains a Radio-frequency identification (RFID) tag that contains information such as the timetable at the bus stop. RFID tags can be read and written by the onboard RFID-reader of most buses and minibuses.

Customers with RFID-enabled phones can tap the poster with their phones to view the timetable. Communities such as schools can even change the location of a bronze stop or create a new stop simply by moving the pole or by attaching a new BainomugiStar poster (containing the logo and an RFID tag) to the wall of a shop, a house, a school gate, etc. A poster (containing the tag) can be bought (and registered) from BainomugiStar.

Golden Stops are stops located in central locations with better digital infrastructure (hotels, shopping malls, and government buildings). Technically speaking, golden stops are equipped with a wireless computing

[1]Kampala is the largest and capital city of Uganda. Thanks to Wolfgang De Meuter for coining the terms *BainomugiStar* and *BainomuAppies*.

Figure 2.1: A *BainomugiStar* poster at a Bronze stop.

device with its own display, just like buses and minibuses are. Golden stops can communicate with the onboard computer of all minibuses and buses that temporarily stop there. Golden stops are entirely owned and maintained by the BainomugiStar Company engineers.

By allowing buses and minibuses to communicate with one another as well as with the digital infrastructure at the Bronze and Golden Stops, BainomugiStar thus operates a fairly cheap infrastructureless urban information network that spans a good part of the Kampala region. Individual passengers that regularly ride a BainomugiStar bus or minibus can obtain a free RFID-tag that is glued on the flip side of their mobile phone. In the BainomugiStar information network, each tag is associated with the personal identity of its owner and other basic information such as shopping and entertainment preferences. As such, a bus or a minibus always knows exactly the customers onboard during some ride. The onboard information system can react accordingly and run the application behaviour that is only relevant for that particular party. Customers who own phones with wireless communication can connect to the onboard information system and interact with the BainomugiStar information network (e.g., injecting new information). BainomugiStar also has a central phone number to which customers without wireless-enabled phones can send SMS-messages. These SMS-messages are then also injected into the information network. BainomugiStar is currently lobbying with the Kampala Capital City Authority to make this central phone number free to all Kampala-registered mobile phone numbers (but not to companies wishing to distribute publicity).

The onboard screen of the digital platform perpetually switches between applications and application behaviours depending on the context. The running applications ("BainomuAppies") and application behaviours switch all

Figure 2.2: BainomuAppies for different physical contexts in Kampala.

the time as the bus drives around, as it approaches (bronze or golden) stops, as it smells other buses, when it is within a range of certain GPS-coordinates and as people get on or off the bus. As such, the currently running applications and application behaviours may be interrupted at any moment. Moreover, when a bus goes back to the previous context, the previously interrupted applications resume running from where they left off. Taken together, the BainomuAppies are a bit like the "flight information" application in a modern airplane; but much more passenger-specific (targeted towards the specific people onboard during a particular ride); much richer in application behaviour that is enriched with context awareness; much more extensible (new BainomuAppies can be added on to cater for new contexts).

Examples of BainomuAppies applications suite are:

BainoInfo is the default application that runs on the onboard screen; that is, the application that is running when no other application takes over. It shows the whereabouts of the bus on a map of Kampala (with a simple GPS mapping on a pre-loaded map); it displays the weather; it shows information about nearby shops and fixed infrastructure such as fuel stations and waste collection or disposal points. This corresponds to the flight information of an airplane.

BainoStickies are bulletin board styled messages that companies and passengers can post on the onboard digital platform, on the infrastructure of a Golden Stop. If the addressee (identified by his RFID tag) gets on board, BainoStickies takes over the screen, and runs the application behaviour that shows the messages. Companies or organisations might even use the BainoStickies system to percolate messages to entire groups of passengers (e.g., for advertisements or public announcements). The currently running advertisements change depending on the contextual information such as the age group of passengers on the

bus. We can think of scenarios where a quorum of addressees has to be onboard in order for a group-based message to take over the screen.

BainoStop is an application that takes over the screen as soon as a bus or a minibus approaches a stop. It shows information about the stop on the onboard screen such as the name of the stop and the timetable. Because bronze stops are very volatile and can be created depending on the need of the BainomugiStar clients, it is impossible for buses or minibuses to have an up-to-date database that maps GPS-coordinates to stops. Rather, the bus or minibus "smells" a stop simply because it approaches the RFID-tag that is attached to the poster. Initially, the tag is empty, but the first time a bus or minibus approaches a stop, the driver can write meta information about the stop on the tag. This is the information that will be shown by BainoStop next time the bus or minibus approaches the stop.

BainoCrimeSpot is an application that shows a map that visualises crime trends in different parts of Kampala. Passengers can view which city neighbourhoods are safe to live in or travel in during the night. When a bus or a minibus reaches an area that is "unsafe" BainoCrimeSpot shows a warning message such that passengers getting off can take extra care. Passengers or local police posts can send crime incidents to the central phone number using SMS or via WiFi for the passengers that own PDAs.

BainoKatale is an application that detects open markets ("obutale") in the neighbourhood and takes over the onboard screen to shows latest information about food prices in the nearby market. Traders can post food prices tagged with the location of the market. When BainoKatale detects that a farmer is onboard it automatically adapts to display the prices for the products that the farmer provides. The farmer can use this information to find attractive offers.

2.3 Characteristics of Reactive Context-aware Applications

Context-aware applications such as the ones described in the above scenario exhibit new characteristics that set them apart from traditional mobile and PC applications. Common to the above scenarios is the notion of *context-constrained executions*, *prompt adaptability* and *sudden interruptibility and resumption*.

Context-constrained executions. Context-aware applications are constrained to run under particular context conditions. This means that the execution of a context-aware application should only proceed when the context conditions are satisfied and should not be allowed to execute in the wrong context situation. This is necessary to ensure that the application behaviour that is presented to the user at any moment in time matches the current context of use. For instance, in the above scenario, it is necessary to ensure that the BainoStop application only runs when the bus is within range of the bus stop and should not be allowed to execute when the bus is out range of the stop.

Prompt adaptability. Context-aware applications need to promptly and continuously adapt their behaviour to match the current context. A single application is composed of variants of behaviours that need to be dynamically adopted depending on the context. As such, the application needs to ensure that when there is a context change the correct variant of the behaviour is promptly adopted without explicit user intervention. For instance, in the scenario described above, the BainoStickies application provides different information suites for different categories of passengers that are currently onboard. As passengers board or leave the bus, BainoStickies promptly and continuously adapts its behaviour to display information that is suitable for the current audience.

Sudden interruptibility. The kind of applications described above, need to be constantly interrupted due to the unpredictable nature of context changes. Unlike traditional applications where a user interacts with a single application from the start to the end of a certain task, unforeseen interruptions are the norm in context-aware applications. If there is a context change, a running application may need to be interrupted such that another application that matches the current context takes over again. For example, in the above scenarios, the application running on the onboard digital platform can change at any moment in time. When the digital platform is running the BainoInfo application and the bus approaches a BainomugiStar stop, the BainoInfo application is interrupted and the BainoStop application takes over. Similarly, when the bus moves out of range of the stop, BainoStop is interrupted immediately and BainoInfo automatically resumes from where it left off.

It should be noted that the above characteristics are not specific to the scenarios presented in this chapter but depict a general pattern that is present in

most context-aware applications. We further discuss other examples of *reactive* context-aware applications in Chapter 8. Even though *reactive* context-aware applications provide end-users with enhanced experiences and richer information, developing such applications remains notoriously difficult.

2.4 Software Engineering Issues

Here we discuss the key software engineering issues that arise when developing *reactive* context-aware applications using the current software technologies.

Context management. Developers of context-aware applications need to deal with context management issues such as low-level context acquisition from physical sensors. Additionally, raw sensor data from such as GPS coordinates need to be aggregated and interpreted into high-level meaningful contextual information such as location names. Moreover, contextual information not only comes from sensors local to a device to but also sensors located on remote devices. This entails dealing with distribution issues (such as context discovery and network disconnections). Also, since sensor data changes all the time, it is necessary to ensure that contextual information needed by a context-aware application is always kept up-to-date. For instance, in the scenario of BainomuAppies in Kampala, the developer must implement the concerns of retrieving the GPS coordinates and interpret them into names of bus stops. Similarly, the developer must implement concerns of retrieving contextual data from the RFID tags associated with phones of the passengers that happen to be on the bus.

Constraining an entire execution to the correct context. Developers of *reactive* context-aware applications must ensure that an entire application's execution happens *only* in the correct context. This issue is made difficult because of the unpredictable nature of context changes. Because context changes can occur at any moment while an application is executing, it is possible that an execution that started in the correct context may end up running in the wrong context. Allowing an application's execution to continue executing in the wrong context may result in incorrect application behaviour. For instance, in the scenario of BainomuAppies in Kampala, suppose the BainoStickies application is running an advertisement for a specified quorum of passengers that are currently on the bus, the advertisements should not be allowed to continue running the moment the quorum is no longer met. Otherwise, allowing the advertisement to continue running when

the quorum is not met would be incorrect. For developers, implementing such concerns is not trivial.

Prompt adaptation of executions. Developers of *reactive* context-aware applications must ensure that all times the appropriate application behaviours are selected for execution. The developer needs to express the logic to enable the application promptly adopt the appropriate behaviour as soon as a new context change is observed, even when such a context change occurs while in the middle of an ongoing application's execution. For a running application to adopt new behaviour, it requires *interrupting* the current execution and starting a new execution that matches the current context. Such concerns are difficult to implement without adequate software technologies. For instance, in the scenario of BainomuAppies in Kampala, as the bus moves about as the bus moves about, applications must adopt appropriate behaviours all the time (e.g., running the BainoKatale as soon as the bus approaches a market adopting the behaviour of the BainoStop application as soon as the bus approaches a bus stop). If such an adaptation occurs while an application is running, then the ongoing execution must be promptly interrupted, in order for the new application behaviour to take over.

Preserving the execution state between interruptions. Developers of *reactive* context-aware applications must ensure that the application's execution state is preserved between interruptions. This is necessary because it is desirable for an application to continue running from the same point it was interrupted when the application goes back to the previous context. For instance, in the scenario of BainomuAppies in Kampala, suppose that the BainoInfo application is running and the bus approaches a bus stop. In that case, the BainoInfo application should be interrupted, its execution state saved, and starting the execution of the BainoStop application. The moment the bus leaves the bus stop, the BainoInfo application should resume from the exact point at which it was interrupted. For the developer, this implies facing the difficulties of manually saving and restoring the entire execution state of the application, which is not trivial.

Ensuring a consistent execution environment. Developers of *reactive* context-aware applications must ensure that an interrupted application's executions resumes in a consistent execution environment. Different applications may be manipulating the same data. As such the currently interrupted executions may be resumed when shared data has been modified by other ex-

ecutions. It is therefore necessary to ensure that modifications to the shared data do not lead to inconsistencies. For instance, in the scenario of BainomuAppies in Kampala, suppose that the BainoInfo applications is interrupted and the BainoKatale application takes over the screen because a farmer happens to get on the bus. Furthermore, suppose that the farmers changes the display language of the digital platform to a desired local language. As soon as the farmer leaves the bus and the BainoInfo resumes executing, the language change that was made for the BainoKatale application should not be visible for the BainoInfo application. Such data management concerns are difficult to implement without appropriate software technologies.

2.5 Middleware vs Programming Languages

The above issues stress the need for a dedicated software technology to facilitate the development of *reactive* context-aware applications. As stated in Chapter 1, the primary goal of this dissertation is design and develop such a software technology. When developing such a software technology, a reoccurring puzzle is whether the software technology should be conceived as either a programming language or a middleware. In this dissertation, we chose a programming language approach over a middleware approach because most of the software engineering issues that we discussed above require a sophisticated software technology to manipulate a program's execution (e.g., interrupting an ongoing execution depending on context). Moreover, using a middleware to interrupt an ongoing program's execution at any moment would require the developer to inject multiple calls to the middleware APIs in the application code. Additional calls to the middleware APIs would also be required in order to save and restore the program's execution state between interruptions. Also, developing such a software technology as a middleware requires a programming language that already provides some support for manipulating a program's execution. For these reasons, this dissertation focusses on developing a novel programming language technology in order to ease the development of *reactive* context-aware applications. As stated in Chapter 1, this choice has also been influenced by the culture of programming language design at the Software Languages Lab of the Vrije Universiteit Brussel, Brussels, Belgium – where this research was conducted. In the next section, we formulate a number of programming language requirements that should be satisfied by such a programming language technology.

2.6 Programming Language Requirements

In Section 2.3, we discussed the characteristics that are exhibited by *reactive* context-aware applications, namely, *context-constrained executions*, *prompt adaptability* and *sudden interruptibility and resumption*. In Section 2.4, we highlighted the software engineering issues that arise when developing *reactive* context-aware applications using the currently available software technologies. These include: *context management, constraining an entire execution to the correct context, prompt adaptation of executions, preserving the execution state between interruptions*, and *ensuring a consistent execution environment*. The complex nature of these issues (e.g., the need to interrupt a program execution at any moment depending on context conditions) stresses the need for a sophisticated software technology to ease the development of *reactive* context-aware applications. In order to tackle these issues we focus on the design and development of a programming language technology.

The first step in designing and developing such a programming language technology is to define the criteria that must be met by a programming language for it to be considered suitable for developing *reactive* context-aware applications. It is therefore the purpose of this section to put forward language requirements for such a programming language technology. We argue that for a programming language to be considered suitable for developing *reactive* context-aware applications, it should satisfy the following requirements: *R*.1 *Chained Context Reactions*, *R*.2 *Context-dependent Interruptions*, *R*.3 *Context-dependent Resumptions*, *R*.4 *Contextual Dispatch*, *R*.5 *Reactive Dispatch*, and *R*.6 *Reactive Scope Management*. We have distilled these requirements from the characteristics exhibited by context-aware applications (cf. Section 2.3) and the software engineering issues that arise when developing *reactive* context-aware applications (cf. Section 2.4).

In the literature of programming language technologies for context-aware applications we found a list of language requirements for context-oriented programming (COP) languages [HCN08, Val11, Gon08]. However, the language requirements for COP are inadequate for *reactive* context-aware applications. We will give a full review of the COP languages and other existing software technologies for context-aware applications in Section 3.2. In the remainder of this section, we describe the programming language requirements for *reactive* context-aware applications.

R.1 Chained Context Reactions

Since the execution of a context-aware application is driven by context changes, it is necessary that the language provides abstractions for repre-

senting context. Typically, context sources are low-level physical sensors such as GPS receivers, accelerometers, and RFID tags. Therefore, a programming language designed for *reactive* context-aware applications should alleviate the developer from the complexities of low-level context management. The programming language should provide abstractions to represent context sources in order for the application to react upon them and adapt its context-dependent behaviour accordingly. Additionally, it should be possible to combine different context sources in order to create new context sources (e.g., combining GPS and accelerometer context sources to create a context source for a user's activity). Because the values of those context sources change over time, the language runtime should ensure that dependent context sources are *always kept up-to-date*. That is, when a context source receives a new value, all the values of the dependent context sources should be automatically updated without requiring the developer to explicitly propagate the context changes across the chain of dependent context sources.

*R.*2 Context-dependent Interruptions

A context-dependent behaviour should be constrained to *run only* under a specified context condition. Typically, context-dependent behavioural variations are represented as procedures in a concrete programming language. Each context-dependent procedure should be associated with a context predicate that determines the correct context the procedure is allowed to execute in. The context predicate should be *implicitly* checked throughout the procedure execution in order to constrain it to the correct context. Traditional ways of constraining a procedure execution by means of conditional expressions are impractical because they require the developer to *explicitly* insert several `if` expressions in the procedure body. Such an approach is too cumbersome and results in programs written in a style where every expression in the procedure body is preceded by an `if` expression. Therefore, a programming language should provide the developer with an appropriate construct to constrain a procedure to a particular context and the language runtime should ensure that the constraint is obeyed throughout the execution. If the context predicate is no longer satisfied while its associated procedure execution is ongoing, then the execution should be interrupted immediately. Therefore, a language runtime should be able to *promptly interrupt* an ongoing procedure execution when its associated context condition is no longer satisfied. This is necessary in order to prevent a procedure from executing in the wrong context, which can lead to incorrect application behaviour. In Figure 2.3 the execution of the procedure `P1` is constrained to run *only* in context `C1`. When the physical context no longer `C1` while the execution of

`P1` is ongoing, the execution is interrupted.

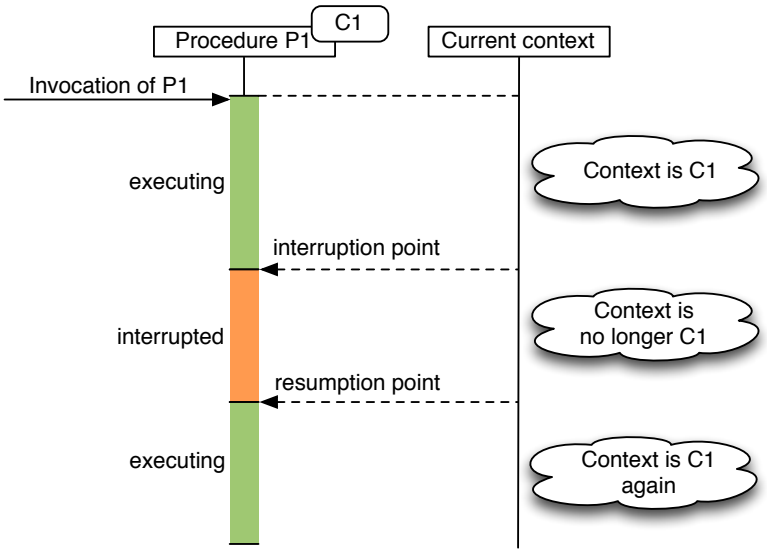

Figure 2.3: Context-dependent interruptions and context-dependent resumptions.

R.3 Context-dependent Resumptions

A programming language designed for *reactive* context-aware applications should support resumption of a previously interrupted execution. Because context changes occur continuously it is possible that a previously unsatisfied context predicate may become satisfied again. When a context predicate that is associated with an previously interrupted execution becomes satisfied again it should be possible to resume the execution from the exact point at which it was interrupted. The resumption of an interrupted execution should be *implicitly* handled by the language runtime and should not require the developer to manually restore a program's execution state. In Figure 2.3 the previously interrupted execution of the procedure `P2` resumes to continue executing from where it left off the moment the physical context becomes `C1` again.

This requirement draws similarities with the unplanned interruptions we as humans are subjected to in our everyday life. Consider, for example, two researchers having a conversation in their office and a third researcher walks into the office. At that moment, the ongoing conversation between the two researchers may be temporarily interrupted and possibly starting a new conversation with the third researcher. As soon as the third researcher

leaves the office, the two researchers can resume their previously interrupted conversation from where they left off. Fortunately for us, the human mind can handle unexpected interruptions and is able to easily resume a previously interrupted task. So, for a language runtime to be able to automatically resume interruptions executions in reaction to context changes, it should be built with support for context-dependent resumptions.

R.4 Contextual Dispatch

A context-aware application consists of definitions of context-dependent behavioural variations for different contexts. Each context-dependent behavioural variation should be associated with a context predicate that determines its applicability. It should be possible to group together related context-dependent behavioural variations under a single grouping entity. Such a grouping entity should enable the developer to dynamically add new context-dependent definitions whenever required without requiring a developer to anticipate all possible future behavioural variations at once. Moreover, the addition of new behavioural variations should not require modification of the existing behavioural definitions.

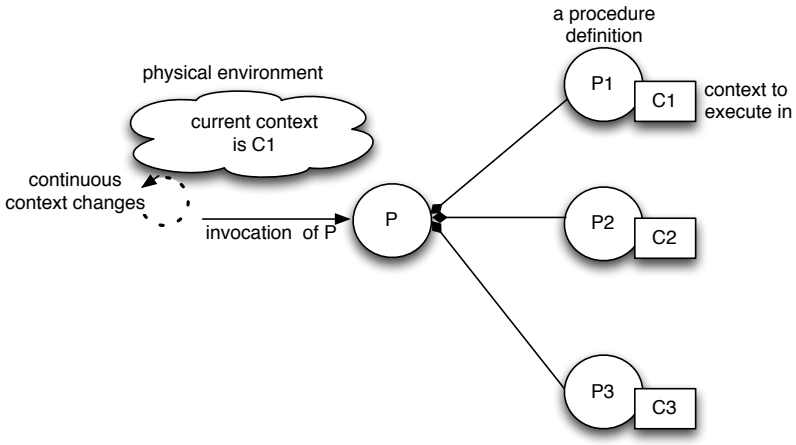

Figure 2.4: Contextual dispatch and reactive dispatch.

The selection of the appropriate context-dependent behavioural variation should be based on contextual information. Given the current context parameters (e.g., the current location or the user's activity) and a set of predicate procedures, the dispatching process should be able to determine which behavioural variation to execute based on the context predicate that evaluates to *reactive*. We refer to this dispatching mechanism as *contextual dispatch*.

Figure 2.4 shows the procedure `P` that has three definitions `P1`, `P2`, and `P3`. Each of the procedure represents a context-dependent behavioural variation for contexts `C1`, `C2`, and `C3`. When `P` is invoked the procedure to execute depends on the current context in the physical environment. Therefore, invoking `P` when the current context in the physical environment `C1` implies that the procedure `P1` is selected for execution.

R.5 Reactive Dispatch

Since context changes occur continuously, it implies that the applicability of a context-dependent behavioural variation to execute depends on a context predicate whose outcome *changes over time*. This implies that a context-dependent behavioural variation that cannot be selected in the current context may eventually become applicable when a context change occurs. Hence it should be selected for execution. This necessitates a sophisticated dispatching mechanism that is repeated in response to context changes. We refer to this kind of dispatching mechanism as *reactive dispatching* in that the dispatcher continuously takes into account any new context changes that occur – even after the first dispatching phase has happened. This in contrast with existing dynamic dispatching mechanisms [MCC98] where the selection of the applicable procedure happens once and is based only on the currently available information. In Figure 2.4 the dispatching process maintains a continuous interaction with the physical environment. If the current context later changes to say `C2`, the dispatching process is repeated and hence the procedure `P2` should be selected for execution.

R.6 Reactive Scope Management

Because an execution of a context-dependent behavioural variation can be suspended at any moment and resumed at a later moment, it is necessary to ensure that the execution is resumed in a consistent environment. For instance, changes to the variables that are shared among context-dependent behavioural variations may become visible to context-dependent behavioural variations other than then one that performed those changes. This can lead to undesirable behaviour (e.g., observing inconsistent values between suspension time and resumption time). It is therefore desirable that a programming language for *reactive* context-aware applications should provide mechanisms to enable the developer to *scope the visibility of state changes*.[2] In Figure 2.5

[2]In this dissertation, we consider state changes caused by assignments. Externally visible side effects such as I/Os are not addressed in this dissertation since they are generally hard to circumvent.

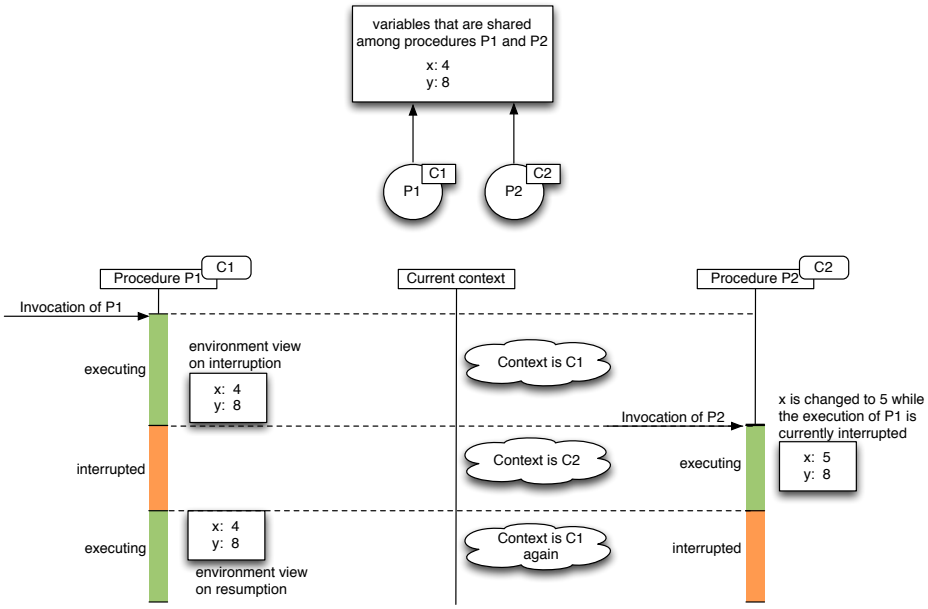

Figure 2.5: Reactive scope management.

procedures `P1` and `P2` share the variables `x` and `y`. When the execution of `P1` is interrupted the value of `x` is `4` while that of `y` is `8`. During the execution of procedure `P2` the value of `x` is changed to `5`. Thus when the execution `P1` is later resumed the value `x` value has been modified to `5` by `P2`, which is different from when it was interrupted. This could lead to incorrect application behaviour. It is therefore necessary to provide language support to scope state changes to the variables that are shared among procedures. In this example, `P1` is resumed with the same environment as when it was suspended. However, a programming language for *reactive* context-aware application may provide different options that the developer can choose from depending on the application.

2.7 Chapter Summary

In this chapter, we have presented the vision of *reactive* context-aware applications where applications continuously sense their environment and *promptly adapt* their behaviour to match the physical context. In order to ground our vision, we presented a visionary scenario called *BainomuAppies in Kampala*. From this scenario, discussed the characteristics that are exhibited by *reactive* context-aware applications, namely, *sudden interruptibility*, *prompt adaptability* and *context-constrained executions*. We subsequently

	Programming Language Requirement
$R.1$	Chained Context Reactions
$R.2$	Context-dependent Interruptions
$R.3$	Context-dependent Resumptions
$R.4$	Contextual Dispatch
$R.5$	Reactive Dispatch
$R.6$	Reactive Scope Management

Table 2.1: Programming language requirements for *reactive* context-aware applications.

discussed the software engineering issues that arise when developing *reactive* context-aware using the currently available software technologies, namely, *context management, constraining an entire execution to the correct context, prompt adaptation of executions, preserving the execution state between interruptions*, and *ensuring a consistent execution environment*. In order to tackle these issues we focus on designing and developing a programming language technology for *reactive* context-aware applications. As a first step towards such a programming language technology, we put forward programming language requirements (cf. Table 2.1), namely, $R.1$ *Chained Context Reactions*, $R.2$ *Context-dependent Interruptions*, $R.3$ *Context-dependent Resumptions*, $R.4$ *Contextual Dispatch*, $R.5$ *Reactive Dispatch*, and $R.6$ *Reactive Scope Management*. These requirements should be satisfied by a programming language in order for it to be considered suitable for developing *reactive* context-aware applications. In the next chapter, we use these language requirements to evaluate previous programming technologies for context-aware applications.

Chapter 3

Programming Technologies for Context-aware Applications: State of the Art

Contents

3.1 Introduction

In Chapter 2, we identified a number of requirements that a programming language should satisfy in order for it to be considered suitable for developing *reactive* context-aware applications. Over the past years researchers have carried out investigations on programming technologies [HCN08, CH05, GMH07, VGC+10, Tan08, KAM11, GCM+11, SGP12b] that aim to ease the development of context-aware applications. The purpose of this chapter is to survey the state of the art of programming technologies and techniques that can facilitate the development of context-aware applications. In this survey, we also cover some programming technologies that were not specifically

designed for context-aware applications but which exhibit properties that map well onto some of the programming language requirements for *reactive* context-aware applications (cf. Section 2.6).

In Section 3.2, we give a review of the programming languages that support *context-oriented programming* for enabling dynamic adaptation of an application's behaviour depending on the context of use. Section 3.3, reviews functional reactive programming languages, which provide reactive abstractions for representing event sources. We also survey advanced control flow constructs and programming language facilities that enable interrupting and/or resuming executions. These include first-class continuations (Section 3.4.1), coroutines (Section 3.4.2), threads (Section 3.4.3), and finally Guards [Lea99], invariants and assertions (Section 3.5.2). We evaluate each approach against the requirements that we put forward in Section 2.6.

3.2 Context-oriented Programming

Context-oriented programming (COP) is a programming language paradigm proposed by Hirschfeld *et. al.* [HCN08] that facilitates developing context-aware applications. The main motivation of COP is to provide programming language support for dynamically adapting an application's behaviour to match the context of use. Hirschfeld *et. al.* identified a number of properties that a COP language should support: (i) *behavioural variations*, (ii) *layers*, (iii) *dynamic activation and deactivation of layers*, and (iv) *scoping of layer activations*. Below, we briefly discuss each of the properties of a COP language as were previously proposed.

Behavioural variations. A COP language should provide support for defining variations of a given behaviour to specify different behaviours for different contexts. Behavioural variations are partial definitions of new or modified behaviour that can be expressed as methods or procedures in the underlying programming language model. Each behavioural variation should be associated with a particular contextual situation.

Layers Layers are entities that group behavioural variations that belong to the same context. Such entities can be first-class and can be referred to in a program.

Dynamic activation and deactivation of layers Layers can be dynamically activated or deactivated depending on the current context of use. Ac-

tivating a layer makes its behavioural variations available while deactivating a layer makes them unavailable.

Scoping of layer activations The scope within which layers are activated or deactivated can be controlled explicitly.

A number of extensions to general purpose programming languages [SGP12a] for COP have been developed. These include ContextL [CH05] for Lisp, ContextS [HCH08] for Smalltalk/Squeak, ContextJ [CHDM06], JCop [App12] and EventCJ [KAM11] for Java, ContextJS [LASH11] for Javascript, ContextScheme [Cos10] for Scheme, ContextPy [HPSA10] and PyContext [vLDN07] for Python. Vallejos and Gonzalez in their respective doctoral theses refined the above COP properties to develop Lambic [Val11] and Ambience [Gon08], respectively. There are also some language extensions that do not fully support the COP properties but which focus on a subset of concerns of context-aware applications. For instance, Context Values [Tan08] is a COP approach that focuses on *context-aware variables* and scoping of side effects. In this section, we review existing COP languages and evaluate them against the programming language requirements for *reactive* context-aware applications that we put forward in Section 2.6.

3.2.1 ContextL and other Layer-based COP Languages

ContextL [CH05] and ContextS [HCH08] were the first concrete language extensions to support the COP properties. Subsequently, other COP language extensions such as ContextJ [CHDM06] and ContextPy [HPSA10] were later implemented based on ContextL and ContextS. We give a detailed review of ContextL only since they are similar.

ContextL is a COP extension to the Common Lisp Object System (CLOS). Like CLOS, ContextL is based on the notion of *generic functions* [BDG+88] where methods belong to a generic function instead of a class.[1] Context-dependent behavioural variations are expressed as different method definitions that belong to a *layered generic function*. Developers can define layered generic functions using the `define-layered-function` construct, which is similar to `defgeneric` for generic function definition in CLOS. Each behavioural variation (method definition) specifies a layer

[1] A generic function is an abstract operation that specifies a name, parameters but without specifying its implementation [Sei04].

it belongs to. For each context there is a corresponding behavioural variation definition. A layer definition represents a particular context. ContextL provides the `deflayer` construct to define such layers. A layer consists of only a name and no further properties of its own, instead ContextL provides constructs that enable developers to add behavioural definitions to them. A layer groups different behavioural variations for the *same* context.

Different behavioural variations are made available or unavailable at runtime through layer activation and deactivation, respectively. Layer activation and deactivation happen *explicitly* by using the `(with-active-layers (`*`layer-name`*`) `*`body`*`)` and `(with-inactive-layers (`*`layer-name`*`) `*`body`*`)` constructs. Activating or deactivating a layer implies that all behavioural variations that belong to that layer become available or unavailable. Such layer activations and deactivations are only in effect within the scope of the construct. Both layer activations and deactivations are *dynamically scoped* meaning that they affect both direct and indirect invocations of the layered generic functions.

We illustrate ContextL's support for context-dependent in Listing 3.1.

Listing 3.1: Expressing context-dependent behaviours in ContextL.

```
1  ;defining layers
2  (deflayer english-layer)
3  (deflayer dutch-layer)
4
5  ;layered generic functions
6  (define-layered-function show-greeting ())
7
8  ;layered method for context-dependent behaviours
9  (define-layered-method show-greeting
10   :in-layer english-layer ()
11   (display "Hello"))
12
13 (define-layered-method show-greeting
14   :in-layer dutch-layer ()
15   (display "Hallo"))
16
17 ;layer activation
18 (with-active-layers (english-layer)
19    (show-greeting))  ===> "Hello"
```

The example displays a greeting message in a different language depending on the user's language (context). We adapt this example from [Tan08].

The code snippet shows the definition of layers `english-layer` and `dutch-layer` for the Dutch and English languages. `show-greeting` is a generic function that is defined using the `define-layered-function` construct. The different behavioural variations for the two contexts are defined as layered methods using the `define-layered-method` construct. Each layered method specifies its containment layer using the `:in-layer` specification. A layer is activated using the `with-active-layers` construct, which is dynamically scoped. In the above example, the `english-layer` is activated. Therefore, invoking `show-greeting` displays the greeting message in English.

Evaluation

We now evaluate ContextL against the requirements of *reactive* context-aware applications that we put forward in Section 2.6. The ✔ and ✖ adjacent to each requirement imply that the requirement is satisfied or not satisfied.

R.1 **Chained Context Reactions** ✖ ContextL and its siblings do not provide any dedicated language abstractions to represent context in a program. A layer is simply a name of the context and does not contain contextual information such as the current location that can be reasoned about in a program. Developers have to provide a representation of context and manually express layer activations and deactivations as reactions to context changes (e.g., by explicitly registering *callbacks* to the context sources).

R.2 **Context-dependent Interruptions** ✖ ContextL and other layer-based COP languages do not provide any language support to constrain a method execution to a particular context. Once a method definition is selected and its execution is started, it is not possible interrupt an ongoing method execution. As such a method execution that started in a correct context may end up executing in the wrong context, which can lead to incorrect application behaviour.

R.3 **Context-dependent Resumptions** ✖ Since ContextL and other layer-based COP languages do not provide support for context-dependent interruptions, they do no support context-dependent resumptions.

R.4 **Contextual Dispatch** ✔ Context-dependent behavioural variations in ContextL and its siblings are represented in terms of method definitions. Each method definition specifies the name of a layer (context

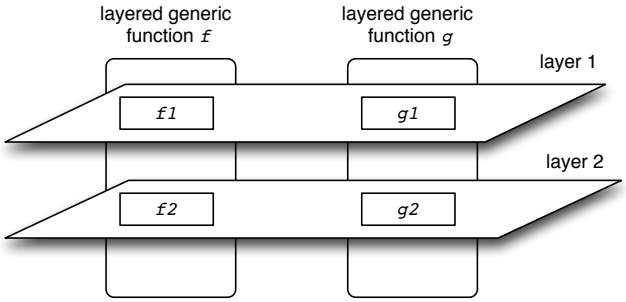

Figure 3.1: Grouping of behavioural variations in ContextL using layers and layered generic functions.

name) it belongs to. As depicted in Figure 3.1, ContextL and other layer-based COP languages provide two ways of grouping together related behavioural variations. The fundamental one to COP being layers, where behavioural variations that belong to the same context are grouped together under the same layer. For instance, in Figure 3.1, behavioural variations `f1` and `g1` belong to `layer 1`. The second grouping mechanism provided by ContextL is layered generic functions, which are similar to the generic functions of CLOS. A layered generic function groups together related behavioural variations for different contexts. For instance, in Figure 3.1, the layered generic function `f` consists of two behavioural variations `f1` and `f2` for different contexts. It is possible to add new behavioural variations at runtime to existing layers or layered generic functions without modifying the existing behavioural variations. Contextual dispatch is achieved by selecting the behavioural variation to execute based on the currently active layers. For instance, if the generic function `f` is invoked when the `layer 1` is active, the behavioural variation `f1` is selected for execution.

R.5 **Reactive Dispatch** ✖ In ContextL and other layer-based COP languages, the selection of applicable behavioural variations happens *only once*. Moreover, the selection is based on the currently active layers, which are *explicitly* activated by the running program.

R.6 **Reactive Scope Management** ✖ In ContextL and other layer-based COP languages, state changes to the variables that are shared among behavioural variations are visible to all behavioural variations that belong to the same layer or generic function. There is no language support that enables the developer to control the visibility of state changes among behavioural variations.

3.2.2 Lambic

Lambic [Val11] is another COP extension to CLOS. Unlike ContextL, Lambic does not have the notion of layers and explicit layer activation. Context-dependent behavioural variations in Lambic are expressed in terms of *predicated generic functions* [VGC+10]. It extends the generic functions model of CLOS to enable method definitions to be associated with context predicates. Context predicates play the role of layers, but context predicates in Lambic can operate on contextual information or state variables. Lambic extends the generic function definition form with the `:predicates` option that enables the developer to specify a list of predicate functions that can be used in the method definitions belonging to that predicated generic function. Similarly, it extends the method definition form with the `:when` option that enables the developer to associate a context predicate with a method definition. Therefore, for each context situation there is method definition with a corresponding context predicate. When a predicated generic function is invoked, the method for execution is selected based on the context predicate that is satisfied.

Listing 3.2 illustrates the definition of context-dependent behaviours in Lambic.

Listing 3.2: Expressing context-dependent behaviours in Lambic.

```
1  ;generic function
2  (defgeneric show-greeting (language)
3    (:predicates english? dutch?))
4
5  ;context-dependent behaviours
6  (defmethod show-greeting (language)
7    (:when (english? language))
8    (display "Hello"))
9
10 (defmethod show-greeting (language)
11   (:when (dutch? language))
12   (display "Hallo"))
13
14 (defvar *language* "English")
15 (show-greeting *language*) ===> "Hello"
```

The example shows the definition of the `show-greeting` generic function using the `defgeneric` construct. It specifies the predicates `english?` and `dutch?`. The predicates return true or false depending on the specified language. Each method definition is associated with a context predicate to determine its specificity. Invoking the `show-greeting` generic function

with `"English"` invokes the appropriate method. Therefore, the greeting message is displayed in English.

Evaluation

We now evaluate Lambic against the language requirements for *reactive* context-aware applications that we put forward in Section 2.6.

R.1 **Chained Context Reactions** ✖ Like ContextL, Lambic does not provide any dedicated language abstractions to represent context sources that depend on physical sensors such as a GPS receiver. As such developers have to deal with the low-level issues of context acquisition as well as manually propagating context changes among dependent context sources in a context-aware application.

R.2 **Context-dependent Interruptions** ✖ In Lambic, context-dependent behavioural variations are expressed in terms of method definitions. Each method definition is associated with a context predicate. However, Lambic does not ensure that the entire method execution is constrained to happen only when the associated context predicate is satisfied. Lambic does not support the interruption of an ongoing method execution even when the associated context predicate is no longer satisfied. Once a method is selected and its execution is started, any context changes that occur during the method execution do not immediately affect the execution (even if they render the context predicate unsatisfied). This may result in a method execution running in a wrong context.

R.3 **Context-dependent Resumptions** ✖ As Lambic does not support context-dependent interruptions, there is no support for resumptions either.

R.4 **Contextual Dispatch** ✔ Lambic like ContextL, inherits the generic functions model of CLOS as its grouping mechanism. Related context-dependent behavioural variations (method definitions) are grouped together under a predicated generic function. New method definitions (behavioural variations) can be added at runtime to an existing predicated generic function without altering the existing method definitions. Lambic employs a *predicate dispatching* mechanism [MCC98] where the method that is selected for execution is based on the context predicate that is satisfied.

R.5 **Reactive Dispatch** ✖ In Lambic the dispatching process happens only once based on the currently available context information. This means it is not possible for future context changes to affect the dispatching process. An exception is raised for cases when there is no satisfied context predicate at the dispatching time (even if some context predicate may later become satisfied).

R.6 **Reactive Scope Management** ✖ Lambic does not provide any abstractions that enable the developer to control visibility of state changes to variables that are shared among behavioural variations. Therefore, any changes to the variables that are shared among behavioural variations become immediately visible to all the behavioural variations sharing those variables.

3.2.3 Ambience

Ambience [Gon08] is a COP language that is built *on top* of Common Lisp. Unlike ContextL and Lambic, Ambience does not rely on the object system of CLOS. Instead, Ambience features its own object system that is based on prototypes rather than classes. Behavioural variations are expressed in terms of *multimethods* [SA05]. A method belongs to one or more prototypical objects on which it is *specialised*.[2] Every method definition is preceded with the `with-context` construct that specifies the context in which the method is applicable. Ambience represents context as first-class objects. There is a predefined root context object from which new context objects can be created through either cloning or delegation-based inheritance. A context object in Ambience plays the role of a layer, but in addition context values can be added as slots of the context object. For every contextual situation (e.g., office or home) there is a corresponding context object. A context object can be activated or deactivated using the respective constructs `activate-context` and `deactivate-context`. In addition, the Ambience runtime includes a dedicated context manager that is responsible for context switching (i.e., activating and deactivating context objects). In Ambience, the activation of a context object is global to the application. The current active object is passed as an implicit argument for each method invocation, therefore, the method that is selected for execution depends on the active context object(s). Recently, an reincarnation of Ambience called

[2] A method is said to be specialised on an object if it specifies that object as one of the arguments it can handle. In Ambience, a method can be specialised on more than one object.

Subjective-C [GCM+11] has been developed as an extension to the Objective-C programming language.

Ambience's constructs for COP are illustrated in Listing 3.3.

Listing 3.3: Expressing context-dependent behaviours in Ambience.

```
1  ;defining context
2  (defcontext english)
3  (defcontext dutch)
4
5  ;defining context-dependent behaviours
6  (with-context english
7    (defmethod show-greeting ()
8      (display "Hello")))
9
10 (with-context dutch
11   (defmethod show-greeting ()
12     (display "Hallo")))
13
14 ;activating a context
15 (activate-context english)
16 (show-greeting) ===> "Hello"
17
18 ;deactivating a context
19 (deactivate-context english)
20
21 ;activating a context
22 (activate-context dutch)
23 (show-greeting) ===> "Hallo"
```

The above example shows the definition of the `english` and `dutch` context objects that are defined using the `defcontext` construct. Methods are defined within the boundaries of the `with-context` construct. The `with-context` construct takes as argument a context object and a method definition. Such a method is implicitly specialised on the specified context object. The method is selected for execution when the context object on which it specialised is active. A context object is activated and deactivated using the `activate-context` and `deactivate-context` constructs respectively. In this example, the `show-greeting` method is invoked after activating the `english` context object. Therefore, the greeting message in English is displayed. Finally, the `english` context object is deactivated and the `dutch` context object is activated. Therefore, the greeting message in Dutch displayed.

Evaluation

We now evaluate Ambience against the requirements of *reactive* context-aware applications that we put forward in Section 2.6.

R.1 **Chained Context Reactions** ✖ Ambience provides a representation of context as a hierarchy of context objects. However, Ambience does not support automatic propagation of context changes among context sources that depend on each other's value.

R.2 **Context-dependent Interruptions** ✖ In Ambience it is not possible to constrain an entire method execution to a particular context. Moreover, there is no support for interrupting an ongoing method execution.

R.3 **Context-dependent Resumptions** ✖ As there is no support for suspending an execution, Ambience does not provide support for resumptions.

R.4 **Contextual Dispatch** ✔ In Ambience, behavioural variations belong to objects on which they are explicitly specialised and the context objects on which they are implicitly specialised. In addition, Ambience permits implicit grouping of multimethods that have the same name but with different specialisers. Method definitions in Ambience are external to the object and it is possible to add new method definitions through the `in-context` construct. Ambience employs the *subjective dispatching* mechanism, where the method that is selected not only depends on the specialisers but also on the current active context object. At the invocation time, every method call has an implicit context object that serves as a context condition that is used to decide the method to execute for the currently active context object.

R.5 **Reactive Dispatch** ✖ The dispatching process happens only once upon each dispatching incident. Moreover, the dispatching process is only is only based on the the current active context objects. Any future context activations cannot affect such a dispatching process unless a running program explicitly invokes the multimethod again.

R.6 **Reactive Scope Management** ✖ In Ambience any state changes to a shared variable is immediately visible to all other methods that share those variables. The language does not provide any dedicated language constructs to enable to developer delimit the scope of state changes.

3.2.4 Contextual Values

Contextual Values [Tan08] is an extension to Scheme that provides support for variables whose values depend on the context in which they are accessed. Unlike previous COP languages that focused on providing different behaviours for different contexts, *Contextual Values* focus on providing language constructs to enable values themselves to be context-dependent. Also, Contextual Values provide support for scoping of side effects (assignments). When a state change is performed on a contextual variable, it only affects the value of the variable that belongs to the context in which the state change was performed. In addition, the author proposes a dedicated language construct `(scoped ctx exprs)` that enables the developer to scope side effects to a certain region of a program execution. The `scoped` construct ensures that all side effects that are performed during the execution of `exprs` remain local to the context `ctx`. The boundary of the `scoped` construct is the dynamic extent of the expressions `exprs`.

Listing 3.4 illustrates Contextual Values using an example.

Listing 3.4: Expressing context-dependent variables in Contextual Values.

```
1  ;context
2  (define language "English")
3
4  ;default contextual value - English
5  (define greeting (make-cv-init (lambda () language) "Hello"))
6
7  ;contextual value for Dutch
8  (set! language "Dutch")
9  (cv-set! greeting "Hallo")
10 (cv-ref greeting) ===> "Hallo"
11
12 (set! language "English")
13 (cv-ref greeting) ===> "Hello"
```

The example shows the definition of the `greeting` contextual variable that is created using the `make-cv-init` construct. It contains different values for different contexts. The values are greeting messages while the contexts are the languages. This example is adapted from [Tan08]. The variable is initialised with the `"Hello"` greeting for English. The greeting message for the Dutch language is initialised using the `cv-set!` construct. Accessing the `greeting` variable when the language is set to Dutch evaluates to `"Hallo"`. Similarly, accessing the `greeting` variable when the language is set to English evaluates to `"Hello"`.

Evaluation

We now evaluate Contextual Values against the programming language requirements for *reactive* context-aware applications that we put forward in Section 2.6.

R.1 **Chained Context Reactions** ✖ Contextual Values do not provide any dedicated language abstractions for representing context. Context information is represented as regular Scheme variables (e.g., as strings).

R.2 **Context-dependent Interruptions** ✖ Since Contextual Values were designed for context-dependent variables, there is no notion of interruption an ongoing procedure execution.

R.3 **Context-dependent Resumptions** ✖ Contextual Values do not support resumption procedure executions since they focus on context-dependent variables.

R.4 **Contextual Dispatch** ✔ Contextual Values support contextual dispatch but for variables. However, one can exploit the first-class status of procedures in Scheme to represent context-dependent behavioural variations with Contextual Values. Hence achieve contextual dispatch for context-dependent behavioural variations.

R.5 **Reactive Dispatch** ✖ In Contextual Values the selection of the value for the current context happens only once.

R.6 **Reactive Scope Management** ✔ The semantics of Contextual Values are designed to control the scope of state changes to only the context in which they are performed. In addition, the author proposes a dedicated language construct to scope side effects to remain local to a certain region of the program execution.

3.2.5 Discussion

In this section, we have reviewed programming language approaches that fall in the category of context-oriented programming. We have evaluated each approach against the requirements for *reactive* context-aware applications that we identified in Section 2.6. From the evaluation, we observe that none of the approaches satisfies the requirements of chained context reactions, context-dependent interruptions, context-dependent resumptions, and reactive dispatch. Most of the layer-based COP languages, such as ContextL

provide support for contextual dispatch. However, there is no support to ensure that the entire execution of a context-dependent behavioural variation is constrained to a particular context. Contextual Values provide some support for contextual dispatch and reactive scope management. However, they do not support chained context reactions, context-dependent interruptions, context-dependent resumptions, and reactive dispatch.

3.3 Functional Reactive Programming

Functional reactive programming (FRP) [WH00, CK06] is a programming paradigm that has recently been proposed for developing applications that continuously react to external changes. We include FRP in this review because FRP languages typically provide reactive abstractions, *behaviours* and *events*, which enable developers to declaratively express programs as reactions to context changes. Behaviours are first-class reactive abstractions for representing time-varying values while events are used for representing streams of timed values. Central to FRP, is the notion of automatic propagation of changes: Functions that operate on behaviours or events are automatically re-evaluated as soon as the value of any of the arguments changes. This means that developers need not worry about the propagation of the new event changes to the rest of the application that depend on them. Elliot et.al [WH00] summarised the key advantages of the FRP paradigm as: *clarity, ease of construction, composability*, and *clean semantics*.

FRP maps well onto context-aware applications because context-aware applications are essentially reactive in nature since they require continuous interaction with their environment. A typical context-aware application involves reacting to external context changes to adapt the application's behaviour to the observed context. Using traditional programming solutions (such as design patterns and event-driven programming) such reactive applications are typically constructed around the notion of asynchronous *callbacks* (event handlers). Unfortunately, coordinating callbacks can be a very daunting task even for advanced developers since numerous isolated code fragments can be manipulating the same data and their order of execution is unpredictable. In the literature, the problem of callback management is infamously known as *Callback Hell* [Edw09]. FRP tackles the above issues by eliminating the use of explicit callbacks. It provides abstractions to express programs as reactions to external events and having the language automatically manage data and computation dependencies.

FRP was first introduced in Fran [WH00], a domain-specific language designed for developing interactive graphics and animations in Haskell. Most re-

cently, FRP ideas have been explored in different language extensions including FrTime [CK06] for Scheme and Flapjax [MGB+09] for JavaScript. In the category of FRP languages, we review FrTime, which can be considered as a representative FRP language. We dub this category *FrTime and its Siblings*. Other FRP languages that fall in this category are Scala.React [MRO10] and AmbientTalk/R [LMVD10].

3.3.1 FrTime and its Siblings

FrTime [CK06] is an FRP extension for Scheme that is designed to ease the development of event-driven applications. It is embodied in the Racket (formerly known as DrScheme) environment [FFP09]. The basic reactive abstractions in the language are *behaviours* and *event streams* to represent continuous time-varying values and discrete values, respectively. To illustrate how to write a reactive program in FrTime consider an example of annotating a geographical map with the current user location as the user moves about. A reactive program to express such an application is as follows.

```
1 ;gps-latitude and gps-longitude are behaviours
2 (map-user-location gps-latitude gps-longitude)
```

At first glance, the above code snippet looks like a regular Scheme procedure application. However, `gps-latitude` and `gps-longitude` are behaviours meaning that they will get new values whenever the user moves about. Every time any of the behaviours gets a new value the `map-user-location` procedure will be automatically re-evaluated. When Scheme procedures are applied to FrTime behaviours, they are *implicitly lifted* to be able to operate on behaviours. Lifting transforms a regular Scheme procedure such that it operates on behaviours. FrTime employs a push-based evaluation model, meaning that the propagation of changes is initiated by the occurrence of new events.

Evaluation

Below, we evaluate FRP against the programming language requirements for *reactive* context-aware applications that we put forward in Section 2.6.

R.1 **Chained Context Reactions** ✔ Even though FRP does not provide support for expressing context-dependent behavioural variations, the *behaviours* and *events* abstractions map well onto the representation of context changes. Context predicates, for instance, can be expressed in terms of behaviours such that they are automatically re-evaluated

without the developer having to register explicit callbacks that react to context changes. Moreover, FRP languages support automatic propagation of changes among dependent behaviours or event sources.

R.2 **Context-dependent Interruptions** ✖ In FRP languages, any change of a behaviour's value always triggers the execution of the dependent function that runs from the start to the end without interruption. There is no support for constraining a function execution to a particular context and no support to interrupt an ongoing execution either.

R.3 **Context-dependent Resumptions** ✖ Since FRP languages do not support context-dependent interruptions, they do not support context-dependent resumptions either.

R.4 **Contextual Dispatch** ✖ Since FRP languages were not designed for context-aware applications, they do not provide dedicate support for contextual dispatch. The developer can define different functions (for different behavioural variations) and parameterise each with appropriate behaviours. However, the lack of support for contextual dispatch implies that the developer has to manually precede each function invocation with an explicit context check.

R.5 **Reactive Dispatch** ✔**(same function re-evaluation)** FRP languages support re-evaluation of the function when any its arguments receive new values. However, it is the same function that is evaluated again.

R.6 **Reactive Scope Management** ✖ FRP languages mostly focus on ensuring efficient propagation of state changes between dependent computations or data. They do not provide dedicated support for controlling the visibility of state changes. However, state changes are not a concern for FRP languages that are embedded in "pure functional languages" such as Haskell.

3.3.2 Reactive SML

Reactive SML [R.98] is reactive library for Standard ML that is based on the notion of *reactive expressions*.[3] Reactive SML was conceived as a low-level framework on top of which reactive programming abstractions can be built. However, the semantics of reactive programming in Reactive SML is closer to the synchronous languages (the precursors of FRP) than the recent

[3] A reactive expression is basically an SML expression.

reactive programming languages. We include it in the review of related work because of its support to suspend and resume an ongoing execution of a reactive expression. A reactive expression can be activated and suspended at certain points determined by the developer. The suspended execution can later be resumed by explicitly activating it again. The library provides constructs `react` and `suspend` to activate and suspend an execution of a reactive expression, respectively. The following code snippet shows an example (adapted from [R.98]) of a reactive expression.

Listing 3.5: Creating a reactive expression

```
1  val exp = rexp (fn () => (print "FIRST\n";
2                            suspend();
3                            print "SECOND\n"))
```

When a reactive expression is activated, it starts executing either until it reaches the end or it encounters a `suspend` invocation, which suspends the execution of the reactive expression. A suspended reactive expression can be resumed by activating it again using the `react` construct. Therefore, the above code snippet will print `FIRST` when it is first activated and `SECOND`, when it is activated again. A reactive expression can explicitly *yield* control to another reactive expression by using the `activate` construct.

Evaluation

The evaluation of Reactive SML mainly focuses on its support for context-dependent interruptions ($R.2$) and context-dependent resumptions ($R.3$).

$R.1$ **Chained Context Reactions** ✖ Reactive SML does not offer any dedicated abstractions for representing context.

$R.2$ **Context-dependent Interruptions** ✔(***explicit***) In Reactive SML, interruptions can be expressed using explicit `suspend`. The developer needs to explicitly insert the `suspend` to suspend an execution. Expressing context-dependent interruptions using this style would imply that each of those constructs should be preceded with a context condition (e.g., in terms of an `if` expression).

$R.3$ **Context-dependent Resumptions** ✔(***explicit***) Reactive SML supports resumption of a suspended execution. However, it must be done explicitly using the `react` construct.

$R.4$ **Contextual Dispatch** ✖ As Reactive SML was not designed for context-aware applications, it does not provide dedicated support for

dispatching based on context. Therefore, the developer has to use explicit checks to select among different functions.

R.5 **Reactive Dispatch** ✖ Reactive SML does not provide support for reactive dispatch.

R.6 **Reactive Scope Management** ✖ Reactive SML does not provide any support for controlling the visibility of state changes. Thus, any state changes made to the variables that are shared among reactive expressions are immediately visible to all the reactive expressions that share those variables.

3.3.3 Discussion

In this section, we have reviewed functional reactive programming languages. Even though functional reactive programming languages were originally not conceived for context-aware applications, their support for behaviours and events for representing event sources, maps well onto some of the requirements for *reactive* context-aware applications. In particular, functional reactive programming languages satisfy the requirements of *chained context reactions* and *reactive dispatch* that we discussed in Section 2.6. However, functional reactive programming languages do not provide support for context-dependent interruptions, context-dependent resumptions, contextual dispatch, or reactive scope management. Reactive SML provides some support to interrupt an ongoing execution and to resume a suspended execution. However, interruptions need to be explicitly expressed by the developer at certain points in the body of a reactive expression. Expressing context-dependent interruptions and context-dependent resumptions using this style is almost impossible as the occurrence of context changes is unpredictable.

3.4 Advanced Control Flow Constructs

Besides the programming languages that are specifically designed for context-aware applications, there are traditional advanced control flow constructs that can also be used to develop *reactive* context-aware applications. These include *first-class continuations* [SDF+09], *coroutines* [Con63], and *threads* [NBF96]. One interesting property of these advanced control flow constructs is that they provide some support for expressing interruptions and resumptions. In this section, we discuss the properties of these advanced control flow constructs and evaluate each of them against the requirements of *reactive* context-aware applications that we put forward in Section 2.6.

3.4.1 First-class Continuations

A *continuation* [SDF+09] is a powerful control flow construct that represents "the rest of a computation at a given point of the computation". Graham [Gra93] describes a continuation as follows.

> "A continuation is a program frozen in action: a single functional object containing the state of a computation. When the object is evaluated, the stored computation is restarted where it left off. [...] A continuation can be understood as a generalisation of a closure. A closure is a function plus pointers to the lexical variables visible at the time it was created. A continuation is a function plus a pointer to the whole stack pending at the time it was created."

Even though continuations were not conceived for developing context-aware applications, their support for capturing a program state and later reinstating it maps well onto the requirements *R*.2 of *Context-dependent Interruptions* and *R*.3 of *Context-dependent Resumptions*. In a way every programming language involves some form of continuations, although in most programming languages continuations are only employed behind the scenes. However, languages like Scheme [SDF+09] and Standard ML [HDM93] expose *full* continuations with a "first-class" status to the developer.

Scheme, for instance, provides the `call-with-current-continuation` construct (commonly abbreviated as `call/cc`). The `(call/cc` *`proc`*`)` construct packages the current continuation as a first-class function and passes it as an argument to the procedure *`proc`*. The continuation function can be stored in a variable, passed around as argument, returned by procedures and invoked later on. When the continuation function is invoked, the computation resumes from where it was when the continuation was captured. In developing *reactive* context-aware applications, continuations can be used to interrupt an ongoing procedure execution when a certain context condition is no longer satisfied, and resume from where it left when the context condition becomes satisfied again at a later moment.

Evaluation

We now evaluate *first-class continuations* against the requirements of *reactive* context-aware applications that we put forward in Section 2.6.

R.1 **Chained Context Reactions** ✖ Continuations were conceived for capturing the execution of a running program. They do not provide abstractions for representing context.

R.2 **Context-dependent Interruptions ✔(*explicit*)** First-class continuations exhibit some properties that can be used to achieve the requirement of the *context-dependent interruptions*. However, programming with continuations (even for experienced developers) is generally perceived as difficult [Gra93]. More so in a context-driven setting where the capturing, saving and resuming of a continuation depends on unpredictable context changes. By capturing the execution state using `call/cc`, the developer is required to explicitly identify the points in the program where an execution state needs to be captured and when it can be interrupted or resumed. Expressing context-dependent interruptions and context-dependent resumptions in this style, implies that the developer has to precede each expression in the procedure body with a conditional expression to check for the validity of a certain context condition. Since a context change can potentially occur at any evaluation step during the execution, it is difficult to determine those interruption points.

R.3 **Context-dependent Resumptions ✔(*explicit*)** With continuations, a suspended execution can be resumed by *explicitly* invoking the saved continuation.

R.4 **Contextual Dispatch ✖** Since continuations were not designed for context-aware applications, they do not provide support for expressing context-dependent behavioural variations and there is no support for contextual dispatch either.

R.5 **Reactive Dispatch ✖** Continuations do not support reactive dispatching of context-dependent behavioural variations.

R.6 **Reactive Scope Management ✖** Although a continuation includes the variables in the lexical environment at the point it was captured, there is no support to control the visibility of state changes that might be performed to those variables. State changes might occur between when a continuation was captured and when the continuation is invoked again (resumed). Consequently, those state changes that were performed to the variables while a continuation was suspended will become visible to the continuation when it is resumed.

3.4.2 Coroutines

Coroutines [Con63], though not designed for context-aware applications, exhibit some properties that can facilitate the development of *reactive* context-

aware applications. The main properties of coroutines as summarised in [Mar80] and recently in [MI09] are the following:

- The state local to a coroutine persists between successive calls.
- The execution of a coroutine is suspended as control leaves it, only to carry on where it left off when control re-enters the coroutine at some later stage.

An interesting observation is the fact that when a running coroutine transfers control to another coroutine it becomes suspended while the execution of the target coroutine is resumed. Most coroutine implementations provide a construct to enable *explicit* transfer of control (typically named `yield` or `resume`). Originally conceived to simplify the implementation of lexical and syntax analysers, coroutines were later adopted as a concurrency construct for providing cooperative-based multitasking [MI09]. Some of the programming languages that support coroutines include Lua [MRI04] and Simula [GOBK79]. As demonstrated in [HFW86] coroutines can easily be implemented using only first-class continuations.

Evaluation

As stated above, coroutines were not designed for developing context-aware applications. However, their support for suspending and resuming of an execution maps well onto the requirements *R*.2 of *Context-dependent Interruptions* and *R*.3 of *Context-dependent Resumptions*. Below, we evaluate coroutines against the requirements of *reactive* context-aware applications that we put forward in Section 2.6.

R.1 **Chained Context Reactions** ✖ The focus of coroutines is on suspending and resuming an execution. They do not provide any dedicated abstractions for representing context.

R.2 **Context-dependent Interruptions** ✔(***explicit***) Coroutines provide support to express interruptions (through suspension) and resumptions. However, in coroutines the developer has to *explicitly* transfer control using the `yield` construct at certain points in the procedure body. In context-aware applications, such control transfers have to happen based on context changes. Developers therefore have to guard the control transfer construct with a context condition. The main difficulty is that in context-aware applications it is not trivial to know at development time at which points in the procedure body to express those concerns. As a consequence, the developer is required to insert multiple

control transfer constructs and guard each one of those with a context condition.

R.3 **Context-dependent Resumptions** ✔(***explicit***) As with the requirement *R*.2 of context-dependent interruptions, resumptions of a suspended coroutine must be done explicitly.

R.4 **Contextual Dispatch** ✖ Coroutines provide no dedicated support to express context-dependent behavioural variations and there is no support for contextual dispatch. The selection of which coroutine to run for the current context needs to be performed manually using conditional expressions.

R.5 **Reactive Dispatch** ✖ As coroutines do not support any form of dispatching, they do not support reactive dispatch either.

R.6 **Reactive Scope Management** ✖ Any state changes that are performed to the variables that are shared among coroutines are immediately visible to all coroutines that share those variables.

3.4.3 Threads

Traditional threads [NBF96, GJSB05] can be used to express executions that can be suspended and resumed. Threads are typically classified according to their scheduling strategy i.e., *preemptive* or *cooperative*[SBS04].

- *Preemptive threads* may be suspended at any moment in order to give a chance to another thread to execute. The decision to suspend a thread is entirely based on the underlying scheduling algorithm and the thread has no control over the transfer of control to another thread. The main drawback with preemptive scheduling is that a thread may be suspended at an inappropriate time, which may result in undesired consequences.

- *Cooperative threads* (also known as green threads) are in charge themselves to explicitly decide when to relinquish control to another thread. Coroutines that we discussed above can also be classified as some form of cooperative threads.

The above description of threads highlights some appealing properties (i.e., the possibility to suspend or resume a running thread) that can be used for *context-dependent interruptions and context-dependent resumptions*. However, the two categories of threads appear at the two extremes. On

the one extreme, there are preemptive threads, which are implicit and their suspension or resumption is entirely based on a preemptive scheduler. Traditional preemptive threads are unsuitable for developing context-aware applications since the interruption of context-aware applications depends on certain context conditions and not on a predefined time-slot by the underlying scheduler. On the other extreme, there are cooperative threads that require one thread to explicitly handover control to another thread. Therefore, cooperative threads seem to be the only viable threading mechanism to context-dependent interruptible executions. However, cooperative threads suffer from the same limitation as coroutines that we discussed in Section 3.4.2.

Evaluation

We now evaluate threads against the requirements of *reactive* context-aware applications that we put forward in Section 2.6. In this evaluation, we will only consider cooperative threads since preemptive threads do not enable one to suspend a thread based on other conditions other than the underlying scheduling algorithm.

R.1 **Chained Context Reactions** ✖ As threads themselves were not designed for context-aware applications, they do not provide any abstractions to represent context.

R.2 **Context-dependent Interruptions** ✔ As with coroutines, using threads for context-dependent interruptions requires the developer to identify the points where an execution may be interrupted. However, in *reactive* context-aware applications this is almost impossible due to the unpredictable nature of context changes. The developer has to face the burden of explicitly inserting several context checks to proceed *every* expression in the procedure body. Moreover, such every context check requires an explicit invocation to suspend the thread. All that burden lies squarely on the shoulders of the developer.

R.3 **Context-dependent Resumptions** ✔ Expressing context-dependent resumptions using threads requires the developer needs to setup a custom management of suspended threads to decide when they need to be resumed.

R.4 **Contextual Dispatch** ✖ Programming languages that support threads typically provide a construct to create a thread that takes as a block of code to execute or a method to run. However, there are no abstractions for defining context-dependent behavioural variations.

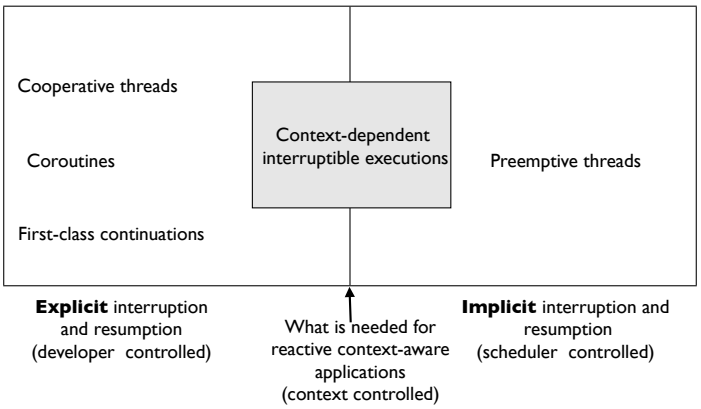

Figure 3.2: An intersection of different programming language facilities for expressing interruptions and resumptions.

Therefore, the developer has to manually select which thread to execute using explicit context checks.

R.5 **Reactive Dispatch** ✖ Threads do not support reactive dispatch.

R.6 **Reactive Scope Management** ✖ Threads are well known as being notoriously difficult to program in the face of shared state [Joh96]. The lack of appropriate abstractions to control the scope of state changes implies that the developer has to employ additional techniques such as *locking* and software transactional memory (STM) [ST95] to deal with side effects that may be performed to the state that is shared among cooperative threads.

3.4.4 Discussion

In this section, we have reviewed existing programming language features that provide some support for interruptions and resumptions. However, none of the approaches in this category supports chained context reactions, contextual dispatch, reactive dispatch and reactive scope management. As depicted in Figure 3.2 first-class continuations, coroutines, and (cooperative) threads can be used to achieve *explicit* interruptions and resumptions. This implies for the developer to manually insert context checks at certain points in a program. However, the unpredictable nature of context changes generated external to the program renders it almost impossible to manually express context-dependent interruptions and context-dependent resumptions. Preemptive

threads can be used to express implicit interruptions and resumptions. However, the interruptions and resumptions in preemptive threads are based on the underling scheduler and not on context changes. What is missing is language support for context-dependent interruptions and context-dependent resumptions that is based on context conditions. Such interruptions and resumptions should be performed implicitly by the language runtime and not by the developer.

3.5 Other Programming Language Facilities

In this section, we review approaches that can be used to express conditions under which a method execution should start or proceed. These include Guards [Lea99], Assertions [Hoa83] and invariants [BEM07]. Approaches in this category also provide some support of what to do when such conditions fail (e.g., blocking or aborting the execution).

3.5.1 Guards

Guards [Lea99] are synchronisation mechanisms that are used to "guard" a method such that it only starts executing under certain conditions. They enable the developer to associate a precondition with a method that specifies whether the execution of the method should begin or not. Such methods are known as *guarded methods* [Lea99]. Before the execution of a guarded method, the condition is checked and if it is false, the execution is blocked, resuming later if and when the condition becomes true. Programming languages that support guards typically provide a `WHEN` or `AWAIT` construct to express such conditionals. For instance, a guarded method can be expressed as follows in a Java extension for Guards.

Listing 3.6: An example of a guarded method in Java

```
1    public void counter ()
2       WHEN (count < MAX) {
3       // method body expressions
4    }
```

The above code snippet shows the definition of the guarded method `counter` whose execution will *only* begin when the condition `(count < MAX)` is true.

Evaluation

We now evaluate Guards against the requirements of *reactive* context-aware applications that we put forward in Section 2.6. As Guards were never conceived for context-aware applications, they do not provide any support for representing context events nor support to dispatch based on the context. However, their support for guarded methods is comparable to some of the requirements for *reactive* context-aware applications, namely, *R*.2 *Context-dependent Interruptions* and *R*.3 *Context-dependent Resumptions*. Below we evaluate Guards against these two requirements.

R.1 **Chained Context Reactions** ✖ Guards do not provide dedicated abstractions for representing context. A condition for a guarded method operates on regular variables and not on context sources such as GPS sensors.

R.2 **Context-dependent Interruptions** ✔**(blocking)** Guards enable the developer to express a guarded method that is associated with a condition that should be satisfied before the execution of the method begins. However, the condition is checked only before the execution of the method and not throughout the execution of the method body expressions. When the guard is not satisfied, the invocation and its associated thread become blocked until the condition becomes true again. Therefore, there is some form of interruption but *only* at the beginning of the method invocation. It is not possible to interrupt an ongoing method execution.

R.3 **Context-dependent Resumptions** ✔**(at the beginning)** Guards provide some support for resumptions but *only* at the beginning of a method execution.

R.4 **Contextual Dispatch** ✖ Guards do not support contextual dispatch. A condition that is associated with a guarded method is *only* used as a precondition for a method invocation but not for the selection of which method to execute for the current context.

R.5 **Reactive Dispatch** ✖ Guards do not support reactive dispatch.

R.6 **Reactive Scope Management** ✖ Guards do not provide support to control the visibility of state changes to variables that are shared among guarded methods.

3.5.2 Assertions and Invariants

Assertions [Hoa83] and invariants [BEM07] found in the *design-by-contract programming* methodology [Mey92] have long been employed as techniques for ensuring program correctness during program execution. They consist of a Boolean expression that is used to determine whether a program execution should proceed or not. This characterisation is similar to the notion of *context-constrained* executions for context-aware applications. With assertions, the developer inserts an *assertion* (a condition) at specific points in a program. If the predicate evaluates to false, the program execution is typically aborted. Invariants, on the other hand, are implicitly checked at the beginning (precondition) of a method execution and at the end (postcondition). An invariant should be satisfied both at the start and at the end, otherwise the execution is aborted.

Evaluation

We now evaluate assertions and invariants against the requirements of *reactive* context-aware applications that we put forward in Section 2.6.

R.1 **Chained Context Reactions** ✖ Assertions and invariants were conceived as software verification mechanisms, therefore, do not provide any dedicated abstractions for representing context events.

R.2 **Context-dependent Interruptions** ✔(***explicit abort***) Assertions and Invariants provide some support for interruptions (abort). With assertions the developer is required to explicitly insert a predicate expression at certain points of the program. With Invariants, the predicate expression is *only* checked at the start and the end of the method body. In both Assertions and Invariants, when the predicate evaluates to false an exception is raised and the execution is aborted.

R.3 **Context-dependent Resumptions** ✖ With Assertions and Invariants it is not possible to resume the interrupted execution later on since executions are aborted on interruption.

R.4 **Contextual Dispatch** ✖ Since Assertions and Invariants have no support for context-dependent behavioural variations, there is no notion of dispatching to select applicable behavioural variation based on the current context.

R.5 **Reactive Dispatch** ✖ Assertions and Invariants do not support reactive dispatch.

R.6 **Reactive Scope Management** ✖ Both Assertions and Invariants do not provide support to control the scope of state changes. Therefore, state changes to variables that are shared among procedures are globally visible even if an execution is aborted due to an unsatisfied precondition or postcondition.

3.5.3 Discussion

In section, we have reviewed programming language facilities that provide some support to constrain a program execution to a particular condition and also provide means to interrupt an execution should that condition not be satisfied or resume the execution should that condition becomes satisfied again. For instance, Guards (cf. Section 3.5.1) enable the developer to express a precondition that should be satisfied before an invocation of a method can begin. Otherwise, the invocation and its associated thread are blocked until the precondition is satisfied. However, a precondition is only checked at the beginning of the method execution. In invariants the condition is checked at the beginning and the end but not throughout the method execution. Assertions require the developer to manually insert conditions at certain points in the method body. Both invariants and assertions provide only abortion as a means of an interruption. They do not provide support for resumptions. None of the approaches in this category supports chained context reactions, contextual dispatch, reactive dispatch, and reactive scope management.

3.6 Synthesis of State of the Art

Table 3.1 presents an overview of the evaluation of the approaches that we have reviewed in this chapter. Each approach is evaluated against the programming language requirements for *reactive* context-aware applications put forward in Section 2.6. As can be seen from the summary of the evaluation, none of the surveyed approaches satisfies all the requirements.

The COP languages provide support for expressing context-dependent behavioural variations, hence they support contextual dispatch. However, in the current COP languages it is not possible to interrupt an ongoing procedure execution. Once a procedure is selected and its execution is started, any context changes that occur during its execution cannot immediately affect the program behaviour. FRP languages offer reactive abstractions to represent events. However, they do not support context-dependent behavioural variations and hence there is no support for contextual dispatch. First-class

	Chained Context Reactions	Context-dependent Interruptions	Context-dependent Resumptions	Contextual Dispatch	Reactive Dispatch	Reactive Scope Management
Context-oriented Programming Languages						
ContextL and other Layer-based COP Languages	✖	✖	✖	✔	✖	✖
Lambic	✖	✖	✖	✔	✖	✖
Ambience	✖	✖	✖	✔	✖	✖
Contextual Values	✖	✖	✖	✔variables	✖	✔`scoped` construct
Functional Reactive Programming Languages						
FrTime and its Siblings	✔behaviours and events	✖	✖	✖	✔the same function re-evaluation	✖
Reactive SML	✖	✔explicit	✔explicit	✖	✖	✖
Advanced Control Flow Constructs						
Coroutines	✖	✔explicit	✔explicit	✖	✖	✖
First-class Continuations	✖	✔explicit	✔explicit	✖	✖	✖
Threads (Cooperative)	✖	✔explicit	✔explicit	✖	✖	✖
Other Programming Language Facilities						
Guards	✖	✔blocking at the beginning	✔from the beginning	✖	✖	✖
Assertions	✖	✔explicit abort	✖	✖	✖	✖
Invariants	✖	✔only at the beginning and the end	✖	✖	✖	✖

Table 3.1: A survey of programming language technologies for context-aware applications.

continuations, coroutines, and (cooperative) threads support *explicit* interruptions and resumptions. However, explicit interruptions and resumptions are not suitable for *reactive* context-aware applications since they require the developer to perform manual context checks throughout the procedure body. Other programming facilities such as invariants, and assertions support limited support of interruption (aborting an execution) and require explicit context checks.

The above observations have motivated the vision of this dissertation for a new programming language model, namely, *interruptible context-dependent executions*, which is the main theme of Chapter 4.

3.7 Chapter Summary

In this chapter, we have reviewed existing programming language approaches that may be candidates for easing the development of context-aware applications. We have classified these approaches into four categories: Context-oriented programming languages, First-class continuations, coroutines, and threads, Functional reactive programming languages, and other programming language facilities, which include Guards, assertions and invariants. Over the past years there have been a number of context-oriented programming languages that are specially designed for context-aware applications. However, none of the current context-oriented programming languages supports all the language requirements for *reactive* context-aware applications. There are existing programming language approaches that provide some support for interruptions and resumptions. However, since those approaches were not designed for context-aware applications, they require the developer to use explicit context checks and do not support the other language requirements for *reactive* context-aware applications. In the next Chapter, we propose a new programming language model that aims to address all the requirements.

Chapter 4

Interruptible Context-dependent Executions

Contents

4.1 Introduction

In Chapter 2, we identified requirements that should be satisfied by a programming language designed for *reactive* context-aware applications. In this chapter, we propose a novel programming language model called *interruptible context-dependent executions* (ICoDE) [BVDR+12] that aims to satisfy those requirements. The primary concept of the ICoDE model is that executions are always constrained to particular context conditions. In this model, executions are automatically interrupted (suspended or aborted) when their

associated context conditions are no longer satisfied. Additionally, suspended executions are resumed or restarted when their associated context conditions later become satisfied again. The goal of this chapter is to discuss the main properties of the ICoDE model. These are: *predicated procedures, representation of context as reactive values, reactive dispatching, interruptible executions, resumable executions,* and *scoped state changes*. For each property, we discuss the aspects to consider when designing a concrete programming language that features the property. In Chapter 6, we present a programming language, *Flute*, which is the first instantiation of the ICoDE model.

4.2 Motivation Revisited

As introduced in Chapters 1 and 2, *reactive* context-aware applications need to *always* be prepared for *sudden interruptions* due to the *unpredictable* occurrence of context changes on the computing device on which they run. Secondly, *reactive* context-aware applications should be able to *promptly adapt* their behaviours to match the current context at any moment during the execution. Additionally, the execution of *reactive* context-aware applications should always be *constrained to a particular prescribed context*. The assumption that a piece of code can run from the start to the end without interruption does not hold for *reactive* context-aware applications. In *reactive* context-aware applications, a context change can occur at any moment during an execution thus requiring a *prompt interruption* of the ongoing execution. When there is a context change, a program execution is expected to "abandon" the ongoing execution, possibly starting a new execution that matches the new context. If the old context later becomes available again, the previously abandoned execution should be reinstated. A challenge that arises is how to express executions that can be interrupted by context changes even if they occur during an ongoing execution. These observations have motivated the ICoDE model that aims to ease the development of *reactive context-aware applications*. A programming language that adheres to the ICoDE model fulfils the requirements put forward in Section 2.6. The subsequent sections present the main properties of the ICoDE model.

4.3 Terminology

Before unveiling the main properties of the model, we will first define the key terms that are used throughout this dissertation.

Definition 1 ***(Context)*** In literature there exist several definitions of con-

text. In [ADB+99] authors define context as "any information that can be used to characterise the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves". The widely cited definition of context in context-oriented programming literature is that "context is any computationally accessible information" [HCN08]. *In this dissertation, we follow the definition of context as in [HCN08] but refine it to refer to computationally accessible information originating from an application's surrounding environment (physical sensors and the user) that can influence the behaviour of the application.*

Definition 2 ***(Context predicate)*** *The term context predicate is used to refer to a function over context parameters that evaluates to true or false.*

Definition 3 ***(Execution)*** *The term execution is used to refer to a running procedure.*

Definition 4 ***(Context-dependent execution)*** *The term* context-dependent execution *is used to refer to an execution that is constrained to run only under a particular context predicate.*

Definition 5 ***(Execution state)*** *The term* execution state *is used to refer to the program counter (i.e., the rest of the expressions to be evaluated) and the bindings that are in the scope of an execution.*

Definition 6 ***(Execution environment)*** *The term* execution environment *is used to refer to the environment that is active at any given position in a program.*

The remainder of this section presents the main properties of the ICoDE model. These are: *predicated procedures, representation of context as reactive values, reactive dispatching, interruptible, resumable executions*, and *scoped state changes*.

4.4 Property #1: Predicated Procedures

A context-dependent program consists of procedure definitions (behavioural variations) for different contexts. Each of those procedures should be associated with a context predicate that specifies the correct context the procedure is constrained to. We refer to a procedure that is associated with a context predicate as a *predicated procedure*. The role of the context predicate is to

ensure that the procedure execution always happens in the correct context. Since a context change can occur at any moment during the procedure execution, it is necessary to check the context predicate throughout the execution of the expressions of the procedure body.

The runtime infrastructure of a language that adheres to the ICoDE model should *implicitly* re-evaluate the context predicate to ensure that the entire procedure execution happens only in the correct context. Without continuous implicit evaluation of context predicates, developers have to manually guard every expression in a procedure body with a context check in order to ensure that the context predicate is respected throughout the procedure execution. Manually guarding expressions is cumbersome and results in programs written in a style where every expression in the procedure body is preceded by a context check. Therefore, a programming language that supports the ICoDE model, should provide a construct to associate a context predicate with a procedure, and the language runtime should ensure that the context predicate is satisfied throughout the procedure execution. The ICoDE's property of *predicated procedures* fulfils the requirement $R.2$ of *context-dependent interruptions* (cf. Section 2.6).

Language Design Considerations for Predicated Procedures

When providing support for predicated procedures in a programming language, there are several design considerations both at the language level and the runtime semantics. For instance, a context predicate can be associated with a procedure either when the procedure is defined or when it is invoked. Another design consideration concerns the choice of the propagation scope of the context predicate (i.e., the extent to which the context predicate should be respected). Additionally, supporting predicate procedures requires considerations on how to group related predicated procedures with mutually disjoint predicates. We discuss these design considerations in the remainder of this section.

DC #1.1: Associating Context Predicates with Procedures

At the language level, we need constructs to define and associate a context predicate with a procedure. This requires a design decision on *when to associate a context predicate with a procedure?* Two options of associating a context predicate with a procedure are discussed below.

As part of the definition. One option is to enable the developer to specify a context predicate as part of the procedure at the definition time. This approach requires different definition styles for context-dependent procedures and regular procedures. However, this change is required only at the definition site and no modifications are required at the invocation site.

Outside the definition. Another option is to disassociate a context predicate from the definition of a procedure. In this case, the language provides a construct to enable the developer to associate a context predicate with an already defined procedure. The obvious gain of this approach is that context-dependent procedures are defined in the same style as regular procedures.

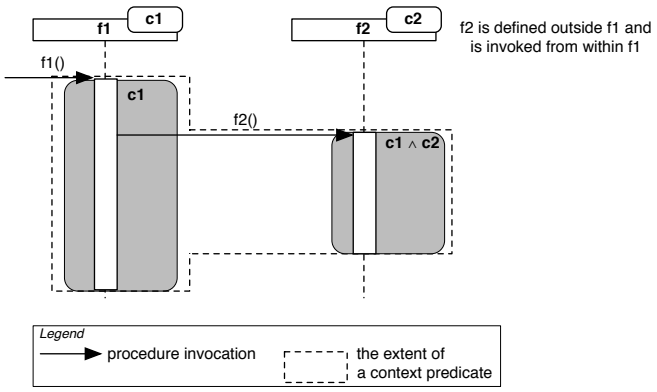

Figure 4.1: Dynamic propagation of context predicates.

DC #1.2: Propagation of Context Predicates

Another design consideration for ICoDE language concerns the propagation of context predicates. Context predicate propagation determines the extent to which a context predicate constrains an execution. For instance, it is necessary to specify whether the context predicate associated with the *caller* procedure is in effect during the execution of the *callee* procedure. We classify the three options of context predicate propagation as *dynamic context predicate propagation*, *lexical context predicate propagation*, and *no context predicate propagation*. Similar characterisations have been used to explain the notion of scope in programming languages [Tan09].

Dynamic Propagation of Context Predicates. In this design option, the associated context predicate that is specified on a procedure is also respected during the execution of the procedures that are directly or indirectly invoked in the procedure body of the *caller*. Conceptually, the context predicates associated with the *caller* and the *callee* are *conjuncted*. This ensures that all context predicate predicates are satisfied; the caller's and the callee's. Figure 4.1 illustrates dynamic propagation of the context predicate *c1*. The procedure *f1* is associated with a context predicate *c1* while *f2* is associated with a context predicate *c2*. Invoking *f2* from within the body of *f1* implies that the execution of *f2* (the *callee*) must also satisfy the context predicate *c1* of *f1* (the *caller*).

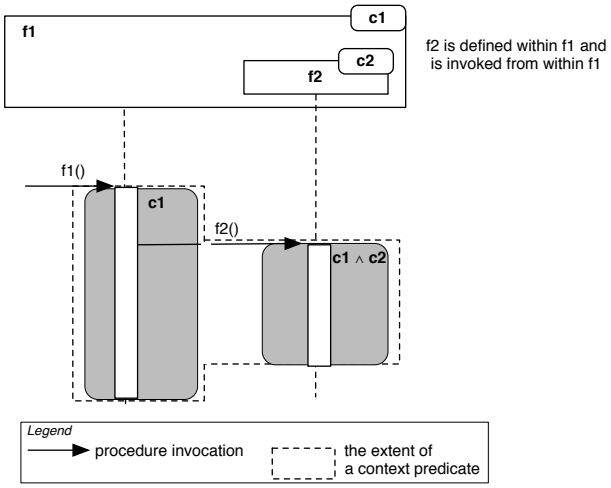

Figure 4.2: Lexical propagation of context predicates.

Lexical Propagation of Context Predicates. With this design option, the context predicate associated with a procedure is also respected during the execution of the procedures that are defined with the lexical scope of the predicated procedure. However, the context predicate is not respected during the execution of the procedures that are invoked from within the body of the predicated procedure. Figure 4.2 illustrates lexical context predicate propagation. The context predicate *c1* of the procedure *f1* is propagated to the procedure *f2* that is defined within the body of *f1*. Procedures that are internally defined within the body of the predicated procedure must also satisfy (in addition to their own context predicate) the context predicate that is associated with the enclosing procedure. This means that all context predicates associated with the predicated procedures in the entire hierarchy of the lexical scope are *conjuncted* and must be satisfied throughout its execution.

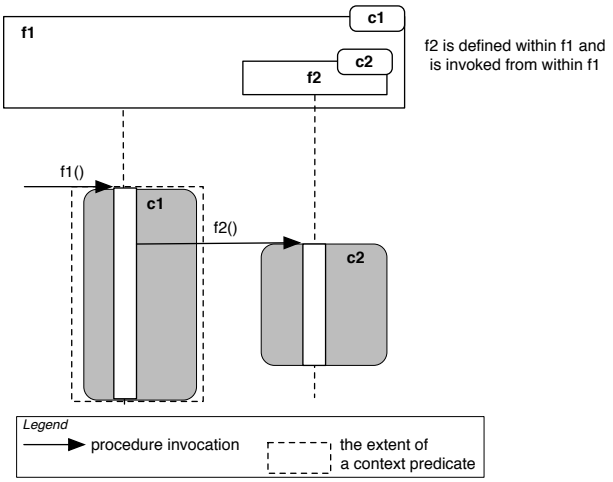

Figure 4.3: No propagation of context predicates.

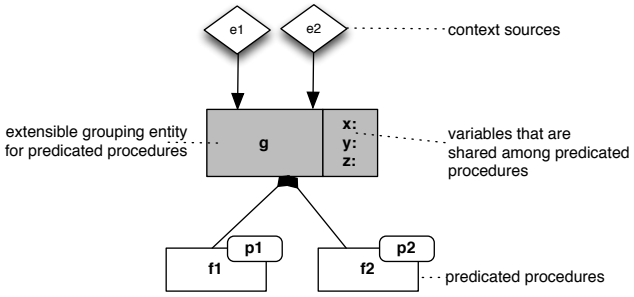

Figure 4.4: An illustration of a group of predicated procedures.

No Propagation of Context Predicates. With this design option, the context predicate that is associated with a procedure is only respected during the execution of that procedure. It is neither respected during the executions of the procedures that are defined nor invoked from within the body of the procedure. Figure 4.3 illustrates the absence of context predicate propagation. The context predicate *c1* of the procedure *f1* is not propagated to the procedure *f2* that is defined and invoked from within the body of *f1*.

DC #1.3: Grouping of Related Predicated Procedures

A context-dependent program consists of a suite of definitions of predicated procedures, each specifying a different behaviour for a particular context. Often those behavioural variants share state variables and their execution is driven by the same context parameters. Therefore, besides constructs for

defining predicated procedures, developers need language support for grouping such context-dependent behavioural variants under a single entity that can be referred to transparently in other parts of the program. That is, it should be possible to initiate the invocation of behavioural variants belonging to the same group by simply invoking their grouping entity. The group entity should specify the variables that are shared among the variants that belong to that entity. Previous COP languages such as ContextL [CH05] and Lambic [Val11] (which we discussed in Section 3.2) use generic functions to group related behavioural variations. It should be possible to add new predicated procedures to an existing grouping entity without requiring modifications to existing predicated procedures. Figure 4.4 shows an illustration of the predicated procedures *f1* (guarded by the context predicate *c1*) and *f2* (associated with a context predicate *c2*) that belong to the grouping entity named *g*. The grouping entity defines shared state variables *x*, *y*, and *z*, and specifies the context sources *e1* and *e2* that influence the behaviour of the predicated procedures. Supporting grouping of related predicated procedures ensures that the requirement *R*.3 *extensible groups of behavioural variations* (cf. Section 2.6) is fulfilled.

4.5 Property #2: Representing Context as Reactive Values

The execution of a context-dependent program is driven by context, which is often obtained from external sources such as GPS sensors. For the developer, this entails dealing with low-level concerns of context representation such as repetitive acquisition and interpretation of raw contextual data. Moreover, when there is a context change, the developer is required to manually propagate such a change among numerous context sources that depend on each other. These issues are further compounded by the unpredictable occurrence of context changes (cf. Section 2.3). As such the execution of a context-dependent program is not driven by a logic specified by the developer but by context changes. Expressing the above concerns using traditional techniques (such as design patterns and event-driven programming) implies the use of asynchronous *callbacks* (event handlers). Unfortunately, manually coordinating callbacks can be a very daunting task since numerous isolated code fragments can be manipulating the same data and their order of execution is unpredictable. To reduce the burden faced by the developers, it is desirable for a programming language that supports the ICoDE model to provide dedicated abstractions for representing context and means to react to context

changes without having to use explicit event handlers.

We refer to the abstraction for representing context sources as *reactive values*. Reactive values are reminiscent of the behaviours (time-varying values) of reactive programming languages [CK06, WH00]. Reactive values are suitable abstractions for context representation because they eliminate the use of explicit callbacks. As such developers can write context predicates in a declarative style. Reactive values represent continually changing values, are first-class and composable abstractions [WH00]. With this abstraction context sources are represented as *reactive values* whose value at any given moment is the current context. Each reactive value implicitly keeps track of all reactive values that depend on its value and ensures that its dependents are automatically updated whenever it observes a context change. In addition, all computations in a program that operate on a reactive value are automatically notified whenever new contextual information is received. By supporting the property of representing context as reactive values, the ICODE model fulfils the requirement $R.1$ *Chained context reactions* (cf. Section 2.6).

4.6 Property #3: Reactive Dispatching

The execution of a context-dependent program requires a dispatching process to determine the appropriate procedures to execute for the current context. Given the current context parameters (e.g., the current location or user preferences) and a set of procedures together with their associated context predicates, the dispatching process should be able to determine which procedure to execute based on the context predicate that evaluates to *true*. The fact that context changes occur continuously, implies that the applicability of a predicated procedure to execute depends on a context predicate whose outcome changes over time. This implies that a predicated procedure that cannot be selected in the current context may eventually become applicable when a context change occurs. This necessitates a dispatching mechanism that is repeated in response to new context changes.

We introduce the concept of *reactive dispatching* where the language's dispatching technique continuously takes into account context changes as they occur – even after a first dispatching phase has happened. In other words, the dispatching process is repeated whenever the values of the contextual parameters change. This implies that when previously unsatisfied context predicates later become satisfied, their associated procedures are selected for execution after all. Similarly, in case there is no satisfied context predicate the dispatching process is repeated later on occurrence of the con-

text changes that render the context predicates satisfied. Additionally, new predicated procedures can be added at runtime and are immediately taken into account by the ongoing dispatching process. This in contrast to existing dynamic dispatching mechanisms [MCC98] where the selection of the applicable procedure happens once and is based only on the currently available information. This property fulfils the requirement *R.4 Reactive dispatch* (cf. Section 2.6).

Language Design Considerations for Reactive Dispatching

Supporting reactive dispatching in a concrete language requires several design considerations. For instance, it is necessary to specify the semantics of dealing with ambiguous context predicates and handling of return values. Below we discuss the language design considerations.

DC #3.1: Dealing with Ambiguous Context Predicates

The selection of the applicable procedure is based on the context predicate that evaluates to true. However, it is possible that more than one context predicate may be satisfied at the same time. This problem is known as *predicate ambiguity problem* [Mil04] in the predicate-based dispatching literature. Previous predicate dispatching approaches solve this issue by verifying that all predicates are mutually exclusive and that at least one predicate must be *true*, at compile time. However, statically verifying the mutual exclusiveness of context predicates is impossible if a programming language model allows developers to write complex predicates. One possible design option is to rely on the developer to always provide mutually exclusive context predicates. Another design option is to consider a language specified choice (e.g., selecting the first one) or the dispatcher can throw an *ambiguous context predicates* exception. In his PhD thesis [Val11], Jorge Vallejos explored the design choice of enabling the developer to specify the order (e.g., by assigning priorities to the context predicates).

DC #3.2: Handling of Return Values

Invoking a group of related predicated procedures may result in several procedure invocations (cf. *DC* #3.1). As such different procedures may finish executing at different times. Therefore, it is necessary to specify how to handle the return values from multiple invocations. Each of the procedures

grouped under the same name may have a return value and therefore, a decision should be taken on how to deal with several return values. We believe that the language should enable the developer to specify how to handle the return values, e.g., merging the values or selecting the first one.

4.7 Property #4: Interruptible Executions

The execution of a predicated procedure should be constrained to happen only under a particular context predicate. This requires that a predicated procedure starts or continues executing only if its context predicate is satisfied. If the context predicate is no longer satisfied while its associated procedure execution is ongoing, then the execution should be promptly "interrupted". A programming language should provide the developer with a number of interruption strategies that specify what to do when the context predicate is no longer satisfied. The choice of the interruption strategy depends on the kind of the task that is expressed by the predicated procedure. By supporting this property, the ICoDE model fulfils the requirement *R*.2 *Context-dependent interruptions* (cf. Section 2.6). We identify two interruption strategies: *aborting the execution* and *suspending the execution*.

Abort. With this interruption strategy, the execution is aborted when the associated context predicate is no longer satisfied. In this case, the execution is aborted and there is no possibility to resume the execution even if a later context change renders the context predicate satisfied again. As a consequence, any state changes to the variables that are shared among executions may need to be undone. We further explore the management of state changes that arise in interruptible executions in Section 4.9.

Suspend. In this interruption strategy, the execution is paused and its execution state is automatically saved when the associated context predicate is no longer satisfied. Pausing an execution means that it is possible to resume the execution later on if its associated context predicate later becomes satisfied again. It is important that such a suspension and the execution state management happens transparently because the unpredictable nature of context changes makes it difficult for the developer to know beforehand when the execution needs to be suspended.

Language Design Considerations for Interruptible Executions

Supporting the property of interruptible executions requires several design considerations. These are concerned with specifying the regions of a procedure body that can be interrupted. We explore two design considerations.

DC #4.1: Implicit Interruptions

In this design choice, a language for ICoDE supports implicit interruptions. This implies that the entire procedure body can be interrupted at any moment during its execution. However, it may be necessary to demarcate certain critical regions of the procedure body that may need to be run without interruption. For instance, in a procedure body, *all-or-nothing* IO actions that should be executed from the start to the end without interruptions. It is therefore, desirable that a language for ICoDE provides a dedicated language construct for the developer to demarcate those critical regions as "uninterruptible".

DC #4.2: Explicit Interruptions

Another design choice is to require the developer to explicitly demarcate certain regions as points of the procedure body that are "interruptible". This means that execution of procedures is by default uninterruptible. The developer needs to identify particular regions as points that can be interrupted by context changes. A drawback with this design choice is that developers have to identify those regions that are subject of interruptions.

4.8 Property #5: Resumable Executions

An execution that has been suspended should be able to resume from where it left off when its associated context predicate later becomes satisfied again. As context changes occur continuously, it is possible that a previously unsatisfied context predicate becomes satisfied again. For instance, a context predicate that depends on the current location may become satisfied or unsatisfied as the user moves about. Such a context predicate may have associated executions that are currently suspended. Therefore, it is desirable for a programming language to enable the developer specify what to do with a previously interrupted execution when its associated context predicate later becomes

satisfied again. For instance, depending on the application, it may be appropriate to resume or restart the execution. By supporting this property, the ICoDE model fulfils the requirement *R.3 Context-dependent resumptions* (cf. Section 2.6). Below we discuss two *resumption strategies* that a programming language can provide to the developer.

Restart. With this strategy, a previously suspended execution is *restarted* from the beginning. This is useful in cases where it is not appropriate to continue the execution from where it was before the context predicate became *unsatisfied*.

Resume. Another strategy is to *resume* a previously suspended execution such that it continues from the exact point where it left off before interruption. This ensures that executions can be seamlessly suspended and resumed depending on the current context of use. The execution state should be restored to the same program instruction when the execution is resumed. Once the execution is resumed, the context predicate should be checked again for the remainder of the execution.

		Resumption strategies	
		Resume	Restart
Interruption strategies	Suspend	✔	✔
	Abort	✖	✔

Table 4.1: Possible combinations of the interruption and resumption strategies.

Note that not all resumption strategies are applicable under every interruption strategy (cf. Section 4.8). For instance, specifying a predicated procedure with the *abort* interruption strategy implies that it is not possible to resume the execution. Therefore, the *resume* resumption strategy is not applicable. Table 4.1 summarises the interaction among the interruption and resumption strategies. The ✔ in the intersection of an interruption strategy and a resumption strategy signifies that the combination is possible. While the ✖ signifies that the combination is not possible.

Language Design Considerations for Resumable Executions

We have so far discussed two possible resumption strategies that determine what to do when the context predicate associated with a suspended execu-

tions becomes satisfied again. However, those strategies do not specify *when* the resumption mechanism is initiated. Since context predicates operate on context parameters, it is inefficient to re-evaluate the context predicates immediately after they have become unsatisfied. This is because the values of the context parameters would most likely not have changed, which means that the context predicate may still be unsatisfied. As a consequence, there would be wasteful checks even when there is no context change that can affect the outcome of the context predicates. Therefore, it is desirable for a language runtime to incorporate a mechanism that continuously monitors context changes and automatically initiates the resumption of the currently interrupted executions when relevant context changes occur. Below we discuss two design considerations.

DC #5.1: Proactive Resumption

One design option is to periodically re-evaluate the context predicate that is associated with the suspended execution. The re-evaluation could be done at every evaluation step or at an interval that is specified by the developer (e.g., every 10 seconds). However, such an approach has drawbacks. For instance, it may lead to wasteful re-evaluation of the context predicate even when no context changes have occurred since the last evaluation. Another drawback with this design option is that it may result in a significant latency between when a context change occurs and when its re-evaluation happens. Hence delayed resumption of the suspended executions. Additionally, context changes may be missed if the change is reverted before the periodic re-evaluation.

DC #5.2: Event-driven Resumption

Another design option is to employ a resumption mechanism that is triggered by the occurrence of relevant context changes. When context parameters receive new values, the context predicates that operate on them are re-evaluated. This in turn may result in suspended executions that are associated with those context predicates to be resumed. This design option eliminates wasteful re-evaluation of context predicates since the re-evaluation only takes place when there is a relevant context change. Moreover, the re-evaluation of the context predicates happens as soon as possible. Event-driven resumption is facilitated by the fact that context is represented as *reactive values* (cf. Property # 2).

4.9 Property #6: Scoped State Changes

The execution of a context-dependent procedure may result in state changes to variables that are shared among context-dependent procedures.[1] As the execution of a context-dependent procedure can be suspended and resumed at a later moment, situations might arise in which state changes performed during the execution of one context-dependent procedure become visible to other procedure executions. This can lead to undesirable behaviour (e.g., observing inconsistent values of a variable between suspension time and resumption time). It is therefore desirable for an ICoDE programming language to provide mechanisms that enable the developer to *scope the visibility of state changes*. By supporting this property, the ICoDE model fulfils the requirement *R.6 Reactive scope management* (cf. Section 2.6). We identify three state management strategies that the developer can select from to scope state changes among executions.

Immediate visibility. Under this strategy, state changes to the variables that are shared among executions are immediately visible by other executions that share this state.

Deferred visibility. Under this strategy, state changes remain local to the execution on interruption but become visible to other executions on completion of the procedure execution. This means that when an execution is interrupted, any state changes performed so far are kept local and are not committed. However, when the execution completes, the state changes are committed and become visible to the rest of the system. In case the procedure's execution is suspended, on resumption the execution continues in the local environment the same as when it was suspended.

Isolated visibility. Under this strategy, state changes remain isolated from the rest of the system. That is, any state changes made by one execution are restricted to that execution and are not visible by other executions. Like in the deferred visibility strategy, a suspended execution resumes in the same local environment as the one it was suspended in. However, the state changes remain local and are not committed even after the execution runs to completion. This ensures isolation of state changes.

Table 4.2 gives an overview of the visibility of state changes for different state scoping strategies at the time the execution is interrupted and at the

[1] In our exploration, we only consider assignments and do not consider external side effects such as I/Os since they are generally hard to circumvent.

		On interruption	On completion
State scoping strategies	*Immediate*	globally visible	globally visible
	Deferred	locally visible	globally visible
	Isolated	locally visible	discarded

Table 4.2: The effect of the state scoping strategies on the visibility of state changes on interruption and completion.

time the execution is completed. In the case of the immediate visibility strategy, state changes are always globally visible on interruption and completion. With the deferred visibility strategy, the state changes are only locally visible on interruption and become globally visible when the execution completes. In the case of the isolated visibility strategy, state changes remain local on interruption and are discarded on completion.

4.10 Chapter Summary

In this Chapter, we have presented a programming language model called *interruptible context-dependent executions* (ICoDE). Table 4.3 summarises the properties of the ICoDE model. For each property we discuss the design considerations that need to be taken into account in order to support it in an ICoDE language. Alongside each property, we show the requirement(s) (cf. Section 2.6) that it satisfies. Below we summarise the properties of the ICoDE model that define the boundaries of a programming language that aims to satisfy the programming language requirements for *reactive* context-aware applications that we put forward in Section 2.6.

- **Property #1: Predicated procedures** for expressing context-dependent behavioural variations. Each context-dependent procedure is associated with a context predicate that is implicitly checked throughout the execution of the procedure body expressions. This property satisfies the requirement of **Context-dependent interruptions** ($R.2$). Related predicated procedures can be grouped together under a single identity. In addition, any variables that are shared among the predicated procedures belonging to the same identity can be specified as part of the group identity definition. New predicated procedures can be added to an existing grouping entity at runtime without requiring any modifications of the existing predicated procedures.

- **Property #2: Representing context as reactive values** makes it possible to compose context predicates with the rest of a context-

Language Property	**Satisfied Requirement**
Property #1: Predicated procedures	
DC #1.1: Associating a context predicate with a procedure	$R.2$ and $R.4$
– As part of the definition	
– Outside the definition	
DC #1.2: Propagation of context predicates	
– Dynamic propagation	
– Lexical propagation	
– No propagation	
DC #1.3: Grouping of related predicated procedures	
Property #2: Representing context as reactive values	$R.1$
Property #3: Reactive dispatching	$R.4$ and $R.5$
DC #3.1: Dealing with ambiguous context predicates	
DC #3.2: Handling of return values	
Property #4: Interruptible executions	$R.2$
Interruption strategies	
–Abort	
–Suspend	
Demarcating interruptible regions	
DC #4.1: Implicit interruptions	
DC #4.2: Explicit interruptions	
Property #5: Resumable executions	$R.3$
Resumption strategies	
–Restart	
–Resume	
Resuming a suspended execution	
DC #5.1: Proactive resumption	
DC #5.2: Event-driven resumption	
Property #6: Scoped State Changes	$R.6$
State scoping strategies	
–Immediate visibility	
–Deferred visibility	
–Isolated visibility	

Table 4.3: Language properties and design considerations for the ICoDE model.

dependent program without having to use explicit event handlers. This property also facilitates event-driven resumption of suspended executions. That is, the resumption suspended executions is event-driven in that previously unsatisfied context predicates are only re-evaluated when relevant reactive values receive new contextual information. This property satisfies the requirement of **Chained context reactions** ($R.1$).

- **Property #3: Reactive dispatching** for selecting which predicated procedure to run for the current context based on the context predicate that evaluates to true. By supporting this kind of dispatching the model satisfies the requirement of **Contextual dispatch** ($R.4$). Moreover, the dispatching process is continuously repeated to take into account of any new context changes. This satisfies the requirement of **Reactive dispatch** ($R.5$).

- **Property #4: Interruptible executions** ensures that the execution of a predicated procedure is constrained to run only under the its prescribed context predicate. If the context predicate is no longer satisfied the execution can be interrupted based on the developer-specified interruption strategies: *suspend* and *abort*. Additionally, the language should provide a construct that enables developers to demarcate certain critical regions in a program as "uninterruptible". This property satisfies the requirement of **Context-dependent interruptions** ($R.2$).

- **Property #5: Resumable executions** ensure that the execution of a previously interrupted procedure execution can be later reinstated. It is desirable that an ICoDE language enables the developer to specify a resumption strategy: *resume* or *restart*. This property satisfies the requirement of **Context-dependent resumptions** ($R.3$).

- **Property #6: Scoped state changes** for controlling the visibility of changes to the variables that are shared among predicated procedures. The model proposes state scoping strategies: *immediate visibility*, *deferred visibility*, and *isolated visibility*. This property satisfies the requirement of **Reactive scope management** ($R.6$).

In Chapter 6, we present a programming language called *Flute* that supports the ICoDE model. By supporting the ICoDE model, Flute satisfies the requirements of *reactive* context-aware applications (cf. Section 2.6). Before introducing the Flute language, the next chapter presents the mobile programming language laboratory that we developed to experiment with the language constructs and features that support the ICoDE model.

Chapter 5

iScheme: A Laboratory for Mobile Programming Languages

Contents

5.1 Introduction

This chapter presents *iScheme*, a language laboratory that we have developed to facilitate experimenting with novel language constructs and features for *reactive* context-aware applications. The goal of iScheme is to provide a language experimentation platform with academic purity while not sacrificing the practical aspects of developing realistic mobile software applications. To accomplish that goal, iScheme blends the rich programming properties of the Scheme language [Ken96] and a state-of-the-art mobile device that is

equipped with context sensors to enable realistic experiments. For our experiments, we chose Apple's iOS devices which include the iPhone smartphone and the iPad tablet. To realise the iScheme mobile language laboratory:

- We port Scheme, which is a small but rich interpreted language, to the iOS platform; a mobile operating system for the iOS devices.
- We engineer a language symbiosis between Scheme and Objective-C (the mainstream language used for iOS development) by way of a reflective API, which facilitates access from within Scheme to the iOS APIs in an event-driven style.
- We design and implement event-driven distribution language constructs specially tailored for distributed mobile computing environments. In particular, the distribution constructs are based on the *ambient-oriented programming model* [Ded06] and have built-in support for peer-to-peer service discovery, asynchronous remote messaging, and failure handling.

The resulting software artefact is a language laboratory that has facilitated our experimentation with the novel language constructs and features that we present in Chapter 6.

5.2 iOS Development As We Know It

As stated in Chapter 2, today's mobile devices are already equipped with context sensors that make it possible to develop realistic context-aware mobile applications. Apple's iOS devices are representative for the modern smartphones and tablets that come with a suite of sensors. The iOS software development kit (SDK) provides rich APIs for interacting with the embedded sensors like an accelerometer, proximity sensor, ambient light, compass, GPS and wireless communication technologies. These APIs facilitate the development of highly dynamic and interactive applications such as location-based services.

Currently, most of the iOS development is restricted to the Objective-C language [Koc09].[1] Objective-C is an object-oriented extension of the C language inspired by the Smalltalk-80 [GL95] syntax. It is a proper superset of C meaning that any valid C program is also a valid Objective-C program. In

[1] While web scripting languages such as JavaScript are an alternative for developing mobile applications, they have very limited access to the underlying native APIs, mainly because of security reasons

fact, some of the iOS APIs are developed in C rather than Objective-C. Taken together, the iOS APIs enable developing realistic mobile applications that exploit hardware capabilities. However, Objective-C has a number of limitations such as the fact that it is quite *low-level*, *limited language extension mechanisms* and the need for *explicit memory management*.[2] These limitations hinder experimenting with new language constructs and features for modern mobile applications. The remainder of this chapter presents iScheme, a language laboratory, that we have developed on top of the iOS APIs to facilitate *practical* experiments with novel language constructs and features for dynamic mobile applications.

5.3 Motivation and the Birth of iScheme

We started the work on iScheme [BVB+12] in 2008, about seven months after the first public release of the iOS SDK (formerly called iPhone SDK) by Apple Inc. At the time, Apple disallowed programming languages other than Objective-C.[3] What started as a mere curiosity to evaluate Scheme expressions on the iPhone device, has evolved into a fully fledged experimentation platform for the novel ideas presented in this dissertation. iScheme integrates the Scheme language[4] [Ken96] and the Objective-C runtime [Koc09].

On the one hand, Scheme brings along the rich programming properties such as *automatic garbage collection, structural macros, and higher-order procedures*. Below we further elaborate on the important language properties that motivated our choice for Scheme.

- Scheme is a small but rich interpreted programming language with a remarkably simple syntax.
- Scheme features *garbage collection*, therefore, the developer does not need to worry about memory management concerns.
- Scheme supports *structural macros*, which is a powerful language extension mechanism that facilitates creating new syntactic constructs. Ad-

[2] While newer versions of Objective-C for the desktop platforms (Mac OS X v10.5 and later) support automatic memory management, the iOS runtime system does not fully support automatic memory management.

[3] However, this restriction was relaxed in 2010.

[4] iScheme is based on a Scheme interpreter, Skem [D'H10], which is developed at the Software Languages Laboratory - Vrije Universiteit Brussel. We chose to use a locally developed Scheme implementation primarily because of the availability of the source code as well as its author – Theo D'Hondt.

ditionally, user-defined constructs are indistinguishable from the primitive ones.

- Scheme supports *higher-order procedures* (i.e., a procedure takes a procedure as argument or returns a procedure as its return value). The support for higher-order procedures maps well onto an event-driven programming model which is the norm in modern mobile applications.

- Scheme provides a dynamic interactive experimentation platform for trying out new ideas or rapid prototyping applications.

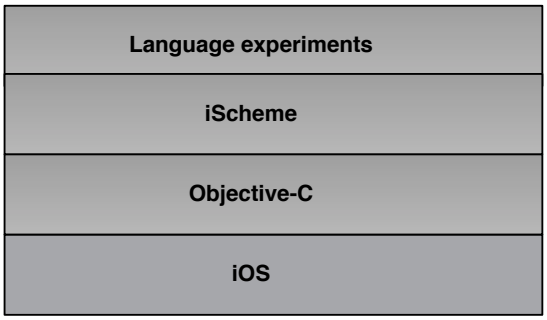

Figure 5.1: An architectural overview of the iScheme language laboratory.

On the other hand, Objective-C and iOS bring along rich libraries such as sensor APIs and the Cocoa framework for graphical user interfaces (GUIs). The integration of these two worlds opens the door for an experimentation platform for new language technologies. With this integration in place, iScheme provides developers with an event-driven programming model for accessing iOS hardware capabilities, with higher-order procedures used as event handlers. In Objective-C, event-driven programs are typically organised around the notion of *delegates*, which serve as callbacks whose methods are invoked when a particular event occurs. Using higher-order procedures as event handlers maps well onto such an event-driven architecture while keeping the simplicity of the Scheme programming model. In addition, previous research [CMB+07] has demonstrated that such an event-driven programming model is also suited for the development of mobile distributed applications. As such, iScheme provides built-in constructs for service discovery (built on top of the Bonjour framework), asynchronous remote messaging (built on top of TCP/IP), and failure handling. This enables distribution concerns to be encapsulated in high-level constructs while relieving developers of the difficulties engendered by distribution.

Figure 5.1 shows an overview of the different components of the iScheme mobile language laboratory. The components of the language laboratory are: the iOS, the Objective-C language and iScheme (on top of which language experiments are conducted). In the next section, we discuss the integration of Scheme and Objective-C.

5.4 Scheme and Objective-C Symbiosis

In order to enable Scheme and Objective-C interaction, we have built a *language symbiosis* layer that is based on the *linguistic symbiosis* model [GWDD06]. The linguistic symbiosis model has been previously used to bridge languages such as SOUL and Smalltalk [Gyb03], AmbientTalk and Java [VCMDM07], and Java and SmallTalk [BDR09]. The linguistic symbiosis model adheres to the following principles:

Data mapping which ensures that data from one language can be passed to another. For instance, when an Objective-C object crosses the boundary to Scheme it needs to be represented as a Scheme value.

Protocol mapping which ensures that one language has a way to invoke another language's behaviour. For instance, Scheme programs require a mechanism to perform message sends to Objective-C objects, while Objective-C programs require a mechanism to call Scheme procedures.

We adhere to the same principles to achieve a language symbiosis between Scheme and Objective-C. However, realising linguistic symbiosis between Scheme and Objective-C is not trivial because of the differences in the programming paradigms. Scheme is based on the procedural programming paradigm where operations are performed by procedure applications, whereas Objective-C is based on the object-oriented programming paradigm where operations are performed by sending messages to objects. As it is not possible to make these differences completely seamless, we provide ways to perform operations from one language to the other. The remainder of this section explains how we achieve language symbiosis between Scheme and Objective-C.

5.4.1 Data Mapping between Scheme and Objective-C

When an Objective-C object crosses the boundary to Scheme, it is wrapped as a Scheme value and, therefore, can be bound to a regular Scheme variable or be passed around as an argument to a Scheme procedure. Figure 5.2 illustrates the representation of an Objective-C object in Scheme. An Objective-C object is wrapped as a generic Scheme value `OBJC_TYPE` that points to

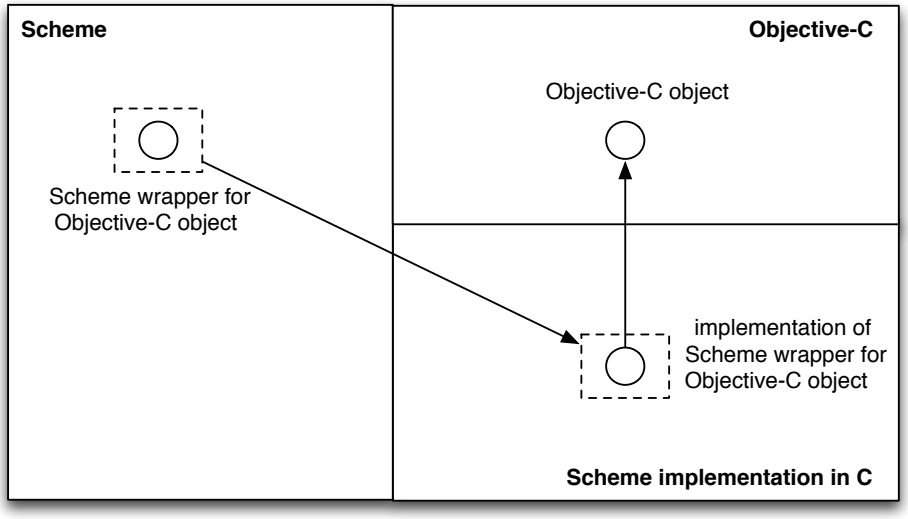

Figure 5.2: Linguistic symbiosis between Scheme and Objective-C.

the actual Objective-C instance. The Scheme interpreter is implemented in C and thus the host language for the Scheme values. Objective-C and C are inherently symbiotic hence no changes are required to host Objective-C objects in C.

When a Scheme value crosses the boundary to Objective-C, it is automatically converted to the corresponding Objective-C type, and it can be bound to an Objective-C variable or passed as argument to an Objective-C method. In addition, we provide type conversion procedures for converting Objective-C values to Scheme values and *vice versa*. For instance, the `string->NSString` procedure converts a Scheme string to an Objective-C `NSString` while the `list->NSMArray` procedure converts a Scheme list to an Objective-C `NSMutableArray`. A complete record of type conversion constructs is provided in Appendix A.1.

5.4.2 Protocol Mapping between Scheme and Objective-C

The difference between the programming paradigms necessitates constructs to enable Scheme and Objective-C to invoke each other's behaviour (procedures or methods). In Scheme, we provide the `OBJC-CLASS`, `OBJC-INSTANCE`, and `OBJC-SEND` special forms to load Objective-C classes, create new objects, and send messages, respectively. Conversely, we provide the `SCHEME_CALL` construct to invoke Scheme procedures from

within Objective-C. The remainder of this section explains these constructs by means of examples.

Instantiating and sending messages to Objective-C objects Scheme programs can instantiate Objective-C classes using the `OBJC-INSTANCE` special form, which performs object allocation and initialisation. In addition, we provide the `OBJC-SEND` special form, which performs message sends to an Objective-C instance. We illustrate these constructs by means of a Scheme procedure that converts a string to audio using the Objective-C class `NSSpeechSynthesizer`. This class implements methods for parsing text and generating synthesised speech.

```
1 (define (speak-out-loud text)
2   (let ((synthesizer (OBJC-INSTANCE NSSpeechSynthesizer)))
3     (OBJC-SEND synthesizer startSpeakingString: text)))
```

The above code snippet shows the definition of the `speak-out-loud` procedure, which implements the behaviour of converting text to audio. The `OBJC-INSTANCE` special form takes as argument a name of an Objective-C class and returns a new instance of the class. The `OBJC-SEND` special form takes as arguments an Objective-C class or instance, the method name, and arguments to the method to be invoked. The `(OBJC-INSTANCE NSSpeechSynthesizer)` expression creates an instance of the `NSSpeechSynthesizer` class and returns a reference to its instance that is subsequently bound to the Scheme variable `synthesizer`. The expression `(OBJC-SEND synthesizer startSpeakingString: text)` invokes the `startSpeakingString:` method on the Objective-C instance stored in the variable `synthesizer` with `text` (the text to speak out) as the argument. The return value of `OBJC-SEND` is a return value of the Objective-C method being invoked and is wrapped as an `OBJC_TYPE` value in Scheme.

Loading Objective-C classes from Scheme The symbiosis layer provides the `OBJC-CLASS` special form which takes as argument a name of an Objective-C class, and returns a first-class reference to its class object. The following example illustrates the use of the `OBJC-CLASS` special form.

```
1 (define NSSynthesizer (OBJC-CLASS NSSpeechSynthesizer))
```

In the above code snippet, the expression `(OBJC-CLASS NSSpeechSynthesizer)` returns a reference to the `NSSpeechSynthesizer` class which gets bound to the Scheme variable `NSSynthesizer`. In case the class name does not exist, the return

value is `nil` (i.e. a null instance).[5] The reference to the class object can be used to invoke class methods. Note that it is also possible to perform object instantiation by sending the `alloc` and `init` messages to the Objective-C class object, but we provide the `OBJC-INSTANCE` special form to this end.

Invoking Scheme procedures from Objective-C Objective-C programs can call Scheme procedures with the `SCHEME_CALL` construct, which takes a Scheme procedure name and a variable number of optional arguments.

```
1 SCHEME_CALL(procedure-name, args)
```

The procedure name can be any globally defined Scheme procedure or a closure that is passed as an argument to an Objective-C method.

5.5 Interacting with Context Sensors in iScheme

Having presented the language symbiosis between Scheme and Objective-C, the stage is now set for reaping the benefits of the resulting language experimentation platform. As introduced in Chapter 2, modern mobile applications require continuous interaction with external sensors and other applications in order to dynamically adapt their behaviour to match the user's needs. For instance a location-based service should maintain a causal connection with the location sensors such that a change in location leads to an immediate adaptation of the behaviour, accordingly. iScheme facilitates developers to easily establish such connections using the symbiosis constructs described above.

To give a feel for how to program in iScheme while interacting with the physical sensors available on the iOS devices, we consider an application that displays a map annotated with the places of interest in the user's neighbourhood. The application works as follows: (1) Using the iOS GPS APIs, the application retrieves the current user's location coordinates. (2) Using a user-specified query that identifies places of interest (e.g., transport stops, restaurants, and bars), the application displays a Google map [Goo09] annotated with the interesting places that are nearby and the user's current location. Figure 5.3 shows the screenshot of the resulting maps application running on the iPhone device.

The listing below shows a complete implementation of the application in iScheme.

[5] In Objective-C, sending a message to `nil` has no runtime effect

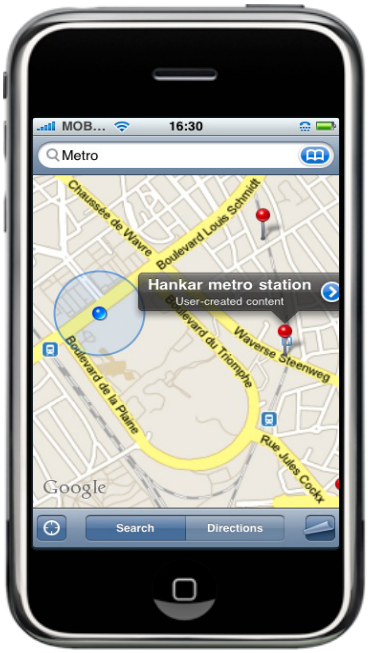

Figure 5.3: `(show-map "Metro")` displays a map annotated with the user's current location and nearby metro stations.

```
1 (define (show-map query)
2   (CURRENT-LOCATION
3    (lambda (latitude longitude)
4      (google-maps latitude longitude query))))
```

The above code shows the definition of the `show-map` procedure that implements the behaviour of the maps application. The `show-map` procedure takes as argument a string query (e.g., "Metro", "Restaurants"). The key part of this program is the `CURRENT-LOCATION` special form that takes as argument a procedure. When the above program is evaluated (e.g., by invoking the `show-map` procedure as `(show-map "Metro")`), the procedure is registered as an event handler for the location changes. The procedure is invoked with the values of latitude and longitude as soon as they become available from the underlying GPS APIs. `google-maps` is a helper procedure that takes as argument the latitude and longitude coordinates and a user query, and displays a Google map annotated with the places matching the user query and the current location.

This simple example already shows the benefits of integrating Scheme with Objective-C. As procedures are first-class values, they can be passed

around as event handlers instead of dealing with the delegate callbacks as is the case in plain Objective-C. The `CURRENT-LOCATION` special form is built on top of the iOS's *Core Location* framework using the symbiosis constructs.

```
1 (define-macro CURRENT-LOCATION
2   (lambda (event-handler)
3     `(let ((Ulocation (OBJC-INSTANCE Ulocation)))
4        (OBJC-SEND Ulocation currentLocation: ,event-handler))))
```

`CURRENT-LOCATION` is defined as a macro that accepts as argument a two-parameter Scheme procedure (an event handler). The two parameters correspond to latitude and longitude location coordinates. `Ulocation` is an Objective-C class that implements a method `currentLocation:` which takes as argument a Scheme procedure that is invoked with the latitude and longitude coordinates as soon as they become available from the GPS receiver. Other constructs for other sources of context information such as the accelerometer and the proximity sensors can be implemented in the same style.

5.6 Ambient-Oriented Programming in iScheme

When developing distributed mobile applications that interact with each other over wireless connections, iOS developers have little more than a low-level socket API to work with. As a result, they have to deal manually with complex distribution concerns such as discovering services in nearby environment, setting up sockets for remote communication, serialising invocations to perform remote method invocations, and handling network disconnections. To alleviate these difficulties, iScheme provides high-level distribution constructs. The distribution constructs in iScheme are based on the *ambient-oriented programming* (AmOP) model [Ded06]. The AmOP model has been instantiated in a number of programming languages including AmbientTalk [CMB+07] and Lambic [Val11].

To that end, iScheme provides distribution constructs for service discovery (built on top of the Bonjour framework), asynchronous remote messaging (built on top of TCP/IP), and failure handling. This enables distribution concerns to be encapsulated in high-level constructs while relieving developers of the difficulties engendered by distribution. More concretely, iScheme provides a reactive event loop distribution model that is based on the event loop model [CMB+07] of the AmbientTalk language. We have built remote

communication around the concept of asynchronous message passing to abstract over network failures without blocking the control flow.

A running example. Throughout this section, we will use a simple news service application as a running example to explain the distribution constructs. In this application, news editors use their mobile devices to submit articles to news publishers as they move about. A news publisher broadcasts current news items, which are printed on the screens of mobile devices of nearby potential customers that have announced their interest in the current news.

5.6.1 Decentralised Service Discovery

As services often need to be discovered in the environment as the user moves about, iScheme offers a built-in publish/subscribe engine to enable applications discover services in a peer-to-peer manner.

Distributed computation in iScheme is expressed in terms of procedures. A procedure represents a certain service offered by a device. A device can acquire a remote reference to a procedure owned by a remote device, and then interact with it through remote procedure invocations. As fixed name servers may not be available when two mobile devices come in communication range and set up a collaboration, iScheme identifies exported procedures by means of service types. A service type is a lightweight classification mechanism used to categorise procedures explicitly by means of a nominal type [CMB+07].

In the example of the news service application, the news publishers need to make available their publishing service to other devices. The code snippet below shows how a developer can explicitly export the procedure representing the news publisher service.

```
1 (define baino-news-type (service-type baino-news))
2 (export-service news-publisher baino-news-type)
```

A service type is defined using the `service-type` procedure. In the above code snippet, the variable `baino-news-type` stores the service type `baino-news`. The `export-service` procedure publishes onto the network a given procedure as the given service type. From the moment a procedure is exported, it is discoverable by procedures residing in other devices by means of its associated service type. In this example, the `news-publisher` procedure is exported on the network as a `baino-news` service. The `export-service` procedure returns a closure that can be used to take the procedure "offline" by invoking the `cancel-publication` procedure.

iScheme employs a publish/subscribe service discovery protocol. A publication corresponds to exporting a procedure by means of a service type (which serves as a "topic" known by both the publisher and the subscriber [EFGK03]). A subscription corresponds to registering an event handler on a service type, which is triggered whenever a procedure exported under that type is encountered on the network. In the news service application, an editor can be notified whenever a news publisher is discovered as follows:

```
1 (when-discovered baino-news-type
2       (lambda (publisher-ref)
3           (submit-news publisher-ref)))
```

The `when-discovered` construct takes as arguments the service type to search for and a one-parameter procedure that serves as an event handler. Such a procedure is invoked with a remote reference to the newly discovered remote procedure associated with that service type. In the above code snippet, whenever a `baino-news` service type is discovered, the `submit-news` procedure is invoked, passing along the parameter `publisher-ref` remote reference received. Similar to the `export-service` construct, the `when-discovered` construct itself returns a closure that can be used to cancel the subscription, by invoking the `cancel-subscription` procedure.

5.6.2 Asynchronous Remote Procedure Invocation

Once a reference to the remote procedure is obtained, remote procedure invocations can be performed by means of the `remote-send!` construct as follows:

```
1 (define (submit-news publisher-ref)
2   (for-each
3    (lambda (article)
4      (remote-send! publisher-ref receive-article article))
5    list-of-articles))
```

The `remote-send!` construct takes as arguments a remote reference, a procedure name, and a variable number of optional arguments. Arguments specified in a remote procedure invocation are parameter passed by copy. Currently only a subset of the Scheme first-class values (Booleans, Numbers, Characters, Symbols, Strings, Pairs and Lists[6]) can be passed in a

[6] Pairs and Lists that contain circular references are not supported in our current serialisation mechanism.

remote procedure invocation. In this example, the `submit-news` procedure iterates over a list storing news articles to be published, and invokes the `receive-article` procedure on the `publish-ref` reference corresponding to the newly discovered news publisher. The `remote-send!` construct performs a non-blocking asynchronous remote procedure call. This means that `remote-send!` enqueues a remote procedure call in the Scheme interpreter and it immediately returns `nil`. As such, callers do not wait for the remote procedure call to be performed remotely nor for the return value of such computation.

In order to get the return value of a remote invocation, we provide the `when-resolved` construct which registers an event handler that is invoked when the return value of the remote procedure invocation becomes available. In our running example, this is used to acknowledge the reception of articles sent to the news publisher.

```
1  (define (submit-news publisher-ref)
2    ....
3    (for-each
4     (lambda (article)
5       (when-resolved
6          (remote-send publisher-ref receive-article article)
7          (lambda (receipt)
8             (set! receipts (cons receipt receipts)))
9          (catch
10            (lambda (exception)
11               ;;exception handling code
12            ))))
13    list-of-articles)
14   ...)
```

The `remote-send` construct is similar to the `remote-send!` construct, except that it returns a *future* (also known as a *promise*) [Hal85, LS88]. A future is a placeholder for the return value that will be computed asynchronously. Once the return value is computed, it replaces the future object, and the future is then said to be *resolved* with the value. Note that registering a future on a `remote-send` construct does not block the caller of the remote procedure call. It is possible to register a block of code which is triggered when the future becomes resolved by means of the `when-resolved` construct. The `when-resolved` construct takes a future and two procedures and registers an event handler on that future. If the future is resolved to a value, the first closure is invoked, passing along the return value of the remote computation. In this example, the `receipt` value is received as the return value of the `receive-article` remote procedure call. If the re-

mote procedure invocation raises an exception, the corresponding future is said to be *ruined* by the exception and the procedure given as an argument to `catch` is is invoked with the exception. This enables applications to catch asynchronously raised exceptions and perform some correcting actions.

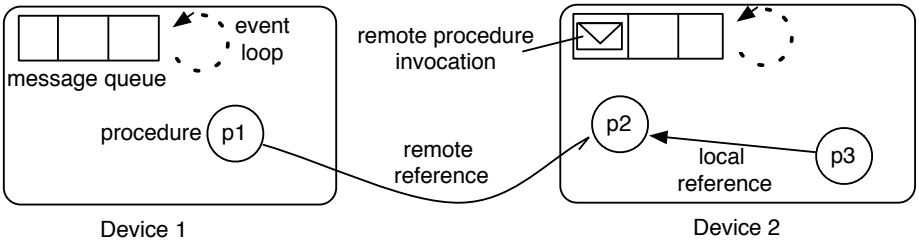

Figure 5.4: A distributed computation model of iScheme.

Performing a remote procedure call using `remote-send` enqueues a remote procedure invocation in iScheme (see Figure 5.4). In case a callee becomes unreachable because of a network failure, the remote procedure invocation remains queued in the caller side. When the network connection is restored at a later point in time, the accumulated procedure invocations are transparently flushed to the remote device in the same order as they were originally performed. However, sometimes disconnections may take unexpectedly longer or devices may not move back into communication range again. To deal with such long-lasting disconnections, developers can associate a *timeout* with the remote procedure invocation to limit the time to wait for the reception of the return value. Such a timeout can be specified by means of the `due-in` construct in the `when-resolved` construct as follows:

```
1  (define (submit-news publisher-ref)
2    .... ;; iterator over each news
3    (when-resolved
4     (remote-send publisher-ref receive-article article)
5     (lambda (receipt)
6       (set! receipts (cons receipt receipts)))
7     (due-in 20.0)
8     (catch
9         (lambda (exception)
10          ;;exception handling code
11          )))
12   ...)
```

The `due-in` construct expects as parameter a number denoting a timeout in seconds. If the return value is not received within the timeout specified,

the future is automatically ruined and the `TimeoutException` is raised which can be handled with the `catch` construct as explained before.

Language construct	**Description**
`OBJC-INSTANCE`	For instantiating an Objective-C class from within Scheme.
`OBJC-CLASS`	For obtaining a reference to an Objective-C class from within Scheme.
`OBJC-SEND`	For performing a message send to an Objective-C instance from within Scheme.
`SCHEME_CALL`	For invoking a Scheme procedure from within Objective-C.
`OBJC_TYPE`	A generic representation of Objective-C values in Scheme.
`CURRENT-LOCATION`	For acquiring GPS coordinates from within Scheme.
`service-type`	For creating a topic (service type) that can be used to discover or publish a procedure.
`export-service`	For publishing a procedure into a network under a specified service type.
`cancel-publication`	For cancelling the publication of a procedure.
`when-discovered`	For discovering a procedure that is available in the network under the specified service type.
`cancel-subscription`	For unsubscribing the subscription.
`remote-send!`	For performing an asynchronous remote procedure invocation with no return value.
`remote-send`	For performing an asynchronous remote procedure invocation with a return value.
`when-resolved`	For handling return values of asynchronous remote procedure invocations.
`due-in`	For specifying the maximum time to wait for return values of an asynchronous remote procedure invocation.
`catch`	For handling exceptions raised during an asynchronous remote procedure invocation.

Table 5.1: An overview of the iScheme language constructs.

5.6.3 Summary

In this section, we have presented the distribution constructs for *ambient-oriented programming* in iScheme. These constructs are built on top of the symbiosis layer using the syntactic extension mechanisms of Scheme (the macro system). Table 5.1 provides an overview of the main iScheme language constructs.[7] In the next section, we demonstrate the use of the all the constructs by using to develop a non-trivial distributed mobile application.

5.7 Case Study: A Distributed Scrabble Game

In this section, we further validate the language symbiosis and distribution constructs by implementing a distributed mobile application using iScheme. We consider an example of a distributed peer-to-peer SCRABBLE®-like game for the iPhone called *AmbiScrabble*. AmbiScrabble is a digital version of a Scrabble-like game where players work collaboratively with their iPhones to form words. Figure 5.5 shows the screenshot of the AmbiScrabble application on the iPhone. A demonstration of the game and other representative applications that we have developed using iScheme can be found on the iScheme website.[8]

Figure 5.5: Screenshot of the AmbiScrabble game running on an iPhone.

[7] A complete record of type conversion constructs is provided in Appendix A.1.
[8] http://soft.vub.ac.be/amop/ischeme/example_applications

5.7.1 Requirements

The requirements of the AmbiScrabble game are as follows:

- Players are organised in teams and each player has a rack of letters. The goal of the game is to consume all the letters of the team by forming valid English words. The team that first consumes all its letters wins.
- Players belonging to the same team can exchange letters among themselves. Also, team members can see each other's letters and a player may request particular letters from another team member.
- The game should be designed in a peer-to-peer fashion without assuming a centralised server to coordinate the game.
- The game should be fault-tolerant such that network disconnections do not hamper the game progress. This requirement is primarily motivated by the fact that the game runs on iPhones equipped with wireless technology. Connectivity using such a technology is often characterised by frequent network disconnections either because of limited connectivity or because users move about.
- We assume that there is only one game being played at a time.

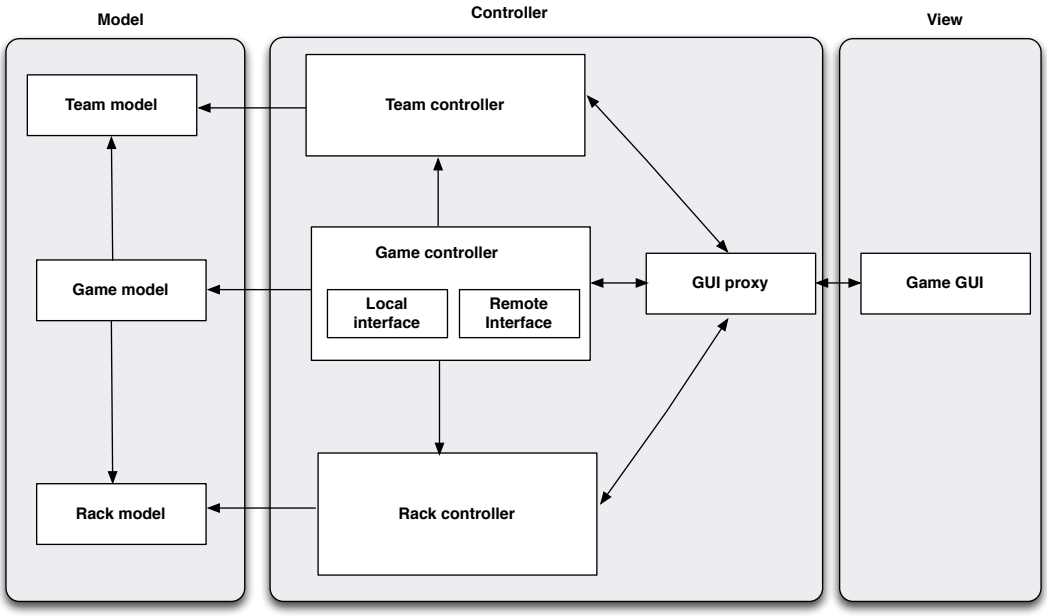

Figure 5.6: Architectural overview of the AmbiScrabble game.

5.7.2 Design and Implementation

The architecture of AmbiScrabble game is based on the Model-View-Controller (MVC) pattern. Figure 5.6 shows the main components of the game. We discuss only representative components of the game for brevity.

- The model holds all the game data and consists of separate model entities: the game model, the team model, and the rack model.
- The controller manages all the game logic and consists of the game controller, the team controller, and the rack controller. The game controller consists of local and remote interfaces for managing to local and distributed interactions, respectively. In addition, the controller includes the GUI proxy that encapsulates all the interactions between the controllers and the graphical user interface.
- The view consists of the game GUI entity which implements the graphical user interface of the game (e.g., virtual letters and racks, teams and players).

We implement the game logic and distribution concerns of the AmbiScrabble application in iScheme, while the graphical user interface (GUI) is implemented in Objective-C using the Cocoa framework. Before discussing the implementation details of the distribution concerns and GUI interactions of the AmbiScrabble game, we give an overview of the game implementation. The following code snippet summarises the relevant parts of the AmbiScrabble application in iScheme.

```
1  (define (create-ambiScrabble-game)
2    (let ((GUI-proxy (setup-gui))
3          (team-controller (make-team-controller))
4          (rack-controller (make-rack-controller)))
5
6      ; the player's local interface procedures
7      (define (add-letter-to-rack letter)
8        (rack-controller 'add-letter letter)
9        (notify-team))
10
11     (define (add-new-player player)
12       (team-controller 'add-new-player player))
13
14     (define (get-player-name) ...)
15     (define (get-player-team) ...)
16
```

```
17      (define (initialise-game team-name player-name)
18        (initialise-player-info team-name player-name)
19        ; engage in peer-to-peer discovery of other players
20        (go-online)
21        (discover-other-players))
22
23      ; the player's remote interface procedures
24      (define (get-player-info)
25        (let ((name (get-player-name))
26              (team (get-player-team)))
27          (list name team)))
28
29      (define (request-letter player-name letter)
30        (process-request player-name letter))
31
32      (define (receive-letter from letters status)
33        (cond ((= status approved)
34               (add-letter-to-rack letters)
35               (notify-team)
36               (alert-request-approved from letters))
37              ((= status refused)
38               (alert-request-refused from letters))
39              ....)
40        'done)
41
42      (define (remote-interface)
43        (lambda (message . args)
44          (case message
45            ((get-player-info)  (get-player-info))
46            ((request-letter)   (apply request-letter args))
47            ((receive-letter)   (apply receive-letter args)))))
48
49      remote-interface))
```

The `create-ambiScrabble-game` procedure consists of: the `GUI-proxy` that manages the game's GUI, the `team-controller` that manages the players and teams in the game, and the `rack-controller` that manages the player's rack and the word formation. The application defines local and remote interfaces which consist of a set of procedures for the local game logic and for interactions with the remote players, respectively. For example, the local interface contains the `add-new-player` procedure that implements the behaviour for adding a player to the appropriate team and the `add-letter-to-rack` procedure that implements the be-

haviour of adding a letter to the player's rack of letters. The remote interface contains the `get-player-info` procedure that is used for obtaining the remote player's name and team, the `request-letter` procedure that is used for requesting a letter from a remote player belonging to the same team, and the `receive-letter` procedure that is used to "throw" letters to the nearby players. The `status` argument in the `receive-letter` procedure indicates whether the request was accepted or not. All the remote interface procedures are wrapped in the `remote-interface` closure that dispatches a remote procedure invocation to the appropriate procedure. As we will explain later the `remote-interface` published in the network in order to make a game instance to become discoverable by other peers.

GUI Interactions

When the `create-ambiScrabble-game` procedure is invoked, the game GUI is launched by invoking the `setup-gui` procedure in the aforementioned `let` construct. The return value of `setup-gui` procedure is a closure that encapsulates all the behaviour of Objective-C and Scheme interaction. The resulting closure is bound to the variable `GUI-proxy`. We show part of the implementation of the `setup-gui` procedure below.

```
1  ; manages all the interactions between Objective-C and Scheme
2  (define (setup-gui)
3    ...
4    ; Create new instance of gameViewController
5    (let ((viewController (OBJC-INSTANCE gameViewController)))
6
7      ; Show new player
8      (define (display-new-player name rack)
9        (let* ((info  (list name rack))
10              (array (list->NSMArray info)))
11         (OBJC-SEND viewController addPlayer: array)))
12
13     ; Remove player from the screen
14     (define (remove-player name)
15       (OBJC-SEND viewController removePlayer: name))
16
17     ; Update team name on the screen
18     (define (set-team-name team-name)
19       (OBJC-SEND viewController updateTeamName: team-name))
20
21     ; Set callback for when letter is selected in word by the user
22     ; Letters should be moved back to the rack.
```

```
23      (define (set-onformword-callback proc)
24        (OBJC-SEND viewController setWordCallback: proc))
25
26      ; Show alert when the formed word is not valid English
27      (define (show-invalid-word-message)
28        (OBJC-SEND viewController showInvalidWordMessage))
29
30      ; Show  a request for a letter on the screen
31      (define (show-request-message player-name letter)
32        (let* ((info  (list player-name letter))
33               (array (list->NSMArray info)))
34          (OBJC-SEND viewController showRequestMessage: array)))
35
36      ...
37
38      ;dispatcher
39      (lambda (message . args)
40        (case message
41          ((set-team-name)          (apply set-team-name args))
42          ((remove-player)          (apply remove-player args))
43          ((show-request-message)   (apply show-request-message args))
44          ...
45          ((display-new-player)     (apply display-new-player args))))))
```

The game's GUI implementation in Objective-C contains the class `gameViewController` that implements methods for capturing user input, and updating the GUI whenever the game data in Scheme changes. In the above code snippet, the variable `viewController` holds a reference to the instance of the `gameViewController` class that is created using `(OBJC-INSTANCE gameViewController)`. The `display-new-player` procedure adds a new player to the GUI and a rack of letters is added on the GUI by invoking the `addPlayer:` method of the `gameViewController` instance. The `remove-player` procedure implements the behaviour of removing a player from the GUI. Procedures that need to be called when a user performs certain actions on the GUI (e.g., a pinch on the submit button to form a word) are registered as callbacks to Objective-C methods. For example, the `set-onformword` procedure registers a Scheme procedure that is called whenever the user presses the button to form word.

Distributed Interactions

When the GUI is launched, the player is prompted to enter a name and a team name. This information is used to initialise the game and the player data by invoking `initialise-game` procedure from Objective-C. The next step of the game setup is to publish the game instance onto the network and to search for other players in the surroundings by invoking the procedures `go-online` and `discover-other-players`, respectively. Note that we assume that there is no dedicated centralised server for game coordination as such. Each game instance publishes and subscribes itself to the network. The following code snippet shows the implementation of the `go-online` procedure that publishes the game.

```
1 ; the service type
2 (define ambiScrabbleService (service-type ambiScrabble))
3
4 ; publishing the game instance on the network.
5 (define (go-online)
6   (export-service remote-interface ambiScrabbleService))
```

A game instance is published onto the network with the `ambiScrabble` service type (stored in the `ambiScrabbleService` variable). More concretely, the `go-online` procedure publishes the `remote-interface` closure as an `ambiSrabble` service using the `export-service` construct.

The `discover-other-players` procedure uses the `when-discovered` construct to register a subscription to discover other players in the network as follows:

```
1 ; discovering other players in the surroundings
2 (define (discover-other-players)
3   (when-discovered ambiScrabbleService
4         (lambda (remote-player)
5           (add-new-player remote-player))))
```

Whenever a new `ambiScrabble` service type is discovered, the `add-new-player` procedure is applied receiving by parameter the newly established reference to the `remote-player` procedure of the remote player. This procedure performs the necessary remote procedure invocation to obtain the remote player's information using the `remote-send` and `when-resolved` constructs as follows:

```
1 (define (add-new-player remote-player)
2   (when-resolved
3    (remote-send remote-player get-player-info)
4    (lambda (info)
```

```
5       (let ((name (list-ref info name-index))
6             (team-name (list-ref info team-index)))
7         (if (team-exists? team)
8             (if (not (player-exists? name))
9                 (add-player remote-player name team-name))
10            (begin
11              (create-new-team team-name)
12              (add-player remote-player name team-name))))))
```

The `add-new-player` procedure performs a remote invocation of the procedure `get-player-info` to retrieve the player's name and team, using the `remote-send` construct. The return value of the remote invocation is received by parameter in the `when-resolved` lambda, which checks whether the team and the player already exist in the data structures of the application (i.e. if the player was previously discovered). If the team already exists but the player does not, then the player is added to that team. Otherwise, the player is added to the newly created team.

To deal with network disconnections of players, the game makes use of the `due-in` construct in the `when-resolved` construct to notify the disconnection of a player. Players whose disconnection exceeds a certain period of time (e.g., 20 seconds) are greyed out in the GUI as shown below:

```
1  (when-resolved
2   (remote-send remote-player get-player-info)
3   (lambda (info) ... )
4   (due-in 20.0)
5   (catch
6       (lambda (exception)
7         (grey-out-player remote-player))))
```

5.7.3 Discussion

In this section, we have shown the language constructs of iScheme by using them to implement a non-trivial distributed mobile application that runs on iPhone devices. The implementation of the distributed interactions of the AmbiScrabble application demonstrate iScheme's distribution language constructs. In addition, the implementation of the GUI interactions of the application demonstrate the language symbiosis constructs. iScheme provides high-level distribution language constructs that facilitate developers to implement distributed mobile applications without dealing with complex low-level concerns of the distributed programming as would be the case in when using plain Objective-C. For instance, the `export-service` construct makes it

possible to easily publish Scheme procedures as services over the iPhone's Wi-Fi connectivity while the `remote-send` construct makes it possible to perform asynchronous remote procedure invocations with support for dealing with network disconnections. The distribution language constructs employ an event-driven programming style with Scheme procedures being used as event-handlers that react to changes in the environment (e.g., appearance of a new service). For instance, the `when-discovered` construct registers a closure as an event-handler that is invoked whenever a new service is encountered in the network.

It should be noted that the distribution language constructs are implemented using the language symbiosis constructs and the language extension mechanism of Scheme. In particular, the Scheme macro system provides support for building syntactic extension constructs. This means that new language constructs can be created on top of the symbiosis constructs to ease the development of mobile applications. One immediate benefit of such an environment is that it serves as an experimentation platform for new programming language abstractions and facilitates prototyping applications without facing the complexities of defining classes and GUIs as is the case in plain Objective-C.

5.8 Implementation Notes

The implementation of iScheme uses a Scheme interpreter called *Skem* [D'H10], that was developed by Theo D'Hondt at the Software Languages Laboratory of the Vrije Universiteit Brussel. Skem is implemented in ANSI-standard C, and it is thus fully compatible with Objective-C. As stated in Section 5.2, Objective-C is a proper superset of C meaning that it is possible to include C code within Objective-C code. In fact, the Objective-C compiler translates every method call in Objective-C into a C procedure call. In iScheme, we use Objective-C 2.0, the modern Objective-C version used by iOS platform and on Mac OS X v10.5 and later.

5.8.1 Reflective Capabilities of Objective-C

Implementing language symbiosis with Objective-C is possible because of Objective-C's dynamism and reflective capabilities. The Objective-C reflective API provides procedures to access information such as the name of a class and the methods that a class implements (introspection). It also enables objects to modify their own structure (intercession) [KR91] such as add new variables, add new classes, or replace method implementations at runtime.

Objective-C is a dynamic language in the sense that messages are not bound to their respective method implementations until runtime. All Objective-C message sends are converted into `objc_msgSend` procedure calls. For every message send expression `[receiver message]` the compiler generates a call on the messaging procedure `objc_msgSend` as follows:

```
1  objc_msgSend(theReceiver, theSelector, arg1....argn);
```

The `objc_msgSend` procedure takes at least two arguments: `theReceiver` which is the receiver object, `theSelector` which is the method name that handles the message and a variable number of arguments for the specified method. The Scheme and Objective-C symbiosis layer has been implemented using the Objective-C reflective API. The distribution constructs are implemented on top of the symbiosis layer. Both the symbiosis and distribution constructs are implemented using the macro system of Scheme.

5.8.2 Limitations

IDE and Debugging Support. iScheme currently comes with a simple front editor as a programming environment for the iPad tablet device (see the screenshot in Appendix A.4). It still lacks enhanced debugging support such as syntax colouring and error reporting.

Performance. The language symbiosis between Scheme and Objective-C introduces a method call overhead compared to method calls in plain Objective-C. For instance, every time an Objective-C object is passed to Scheme needs to be wrapped as a Scheme generic value `OBJC_TYPE`. In addition, for each Scheme value passed to an Objective-C method, the symbiosis layer needs to check and perform a type conversion to the appropriate Objective-C object. This can be improved by applying performance enhancing techniques such as caching of common used objects and selectors.

Serialisation. The current iScheme implementation has a limited serialisation mechanism of datatypes that are parameter-passed to a remote procedure. For example, there is no support for serialising all first-class values in Scheme such as closures and continuations.

5.9 Related Work Notes

In this section, we revisit some of the existing work on Scheme implementations for mobile platforms and language symbiosis with either Objective-C or Scheme.

5.9.1 Scheme Implementations on Mobile Platforms

The Gambit-C Scheme system has been recently used to develop iOS applications [Fee09]. Gambit-C compiles Scheme programs to C code, which is compatible with Objective-C. The Gambit-C system also includes an interpreter that can used to provide an interactive environment. To interact with Objective-C, Gambit-C employs the foreign procedure interface (FFI) approach. FFIs in Gambit-C support interaction in either direction, and as such it qualifies as a language symbiosis. To access Objective-C methods from Scheme, wrapper C procedures are generated. The need to generate wrapper procedures in C for Objective-C methods could be eliminated by adopting Objective-C's reflective API that is used in iScheme. The use of the reflective API also implies that Scheme programs have direct interaction with Objective-C which facilitates the implementation of event-driven constructs (e.g., for accessing iOS capabilities and distributed programming).

Moby Scheme is an experimental Scheme compiler for smartphones with particular target for the Android OS [Kri09]. It enables developers to write Scheme programs that are fed to the compiler in order to generate Java source code. The Moby Scheme compiler is mostly written in PLT Scheme [FFP09] – a Scheme implementation that is designed to run on traditional desktop platforms and not on mobile devices. The Moby Scheme compiler itself does not run on Android OS, but it generates Java source code from the desktop platform. The generated Java source code is used to produce the Android `.apk` packages that can take advantage of the native features (e.g., GPS and SMS) available on the device. One important feature of Moby Scheme is its support for event-driven programming based on the notion of "Worlds" [FFK10] that enables one to write programs that react to events (e.g., an incoming SMS event). As in Moby Scheme, we believe that the event-driven programming style is well-suited for a mobile setting because of their inherent interactive nature. In iScheme, we further explore the event-driven programming model for developing distributed applications by providing constructs for peer-to-peer service discovery, asynchronous remote communication, and handling network failures.

5.9.2 Language Symbiosis

There exist systems that explore language integration with either Objective-C or Scheme, but on desktop platforms. In this section, review some of those systems.

Language symbiosis with Objective-C. Java and Objective-C symbiosis [App09] is one of the earliest bridges to Objective-C that enables one to write programs in Java that instantiate and use Objective-C classes from Java, pass Java objects as arguments to Objective-C methods and directly subclass Objective-C classes. PyObjc[PyO09] implements language symbiosis between Python and Objective-C that enables Python developers to write Cocoa GUI applications on Mac OS X in pure Python. CL-ObjC [Geo09] is a Common Lisp library that enables interaction with Objective-C libraries built on top of a foreign procedure interface. CL-ObjC provides the `invoke` construct similar to our `OBJC-SEND` to perform message sends to Objective-C instances in a LISP-like way. In addition, CL-OBJC implements an interface that mixes the Common Lisp Object System (CLOS) and Objective-C's object system. Unfortunately, all these existing language symbiosis to Objective-C are limited to desktop development platforms (mostly Mac OS X) and no single implementation ports to a mobile platform. Our Scheme and Objective-C symbiosis mainly aims at providing access to features specific to mobile platforms such as phone, SMS and GPS.

Dot-Scheme is a library that builds an FFI for PLT Scheme [FFP09] with Microsoft .NET Common Language Runtime (CLR) [Ped03]. It provides the `import-assembly` construct that loads the assembly code of a class into Scheme. For each loaded class a Scheme proxy is generated that wraps the class as a Scheme value. Dot-Scheme enables invoking the CLR methods using the Scheme-like syntax though it does not provide support to invoke Scheme procedures from the CLR.

Kawa and SILK are Scheme implementations that enable interaction between Scheme and Java. Kawa [Bot98] provides procedures to invoke Java methods from Scheme. The `invoke` construct in Kawa is similar to `OBJC-SEND` construct in iScheme. In addition, Kawa provides different variants of `invoke`, namely, `invoke-static` to call static methods, and `invoke-special` to call only methods in the super class. SILK [AH00] provides constructs `import` and `class` to load Java packages and classes, respectively. The `class` construct in SILK is similar to the `OBJC-CLASS` construct in our iScheme. Surprisingly, neither Kawa nor SILK provides

syntactic sugar for Scheme/Java interaction. For instance, in Kawa the Java method name argument for `invoke` construct must be a string or a symbol. SILK requires the developer to specify the Java package or the class to be imported as a string. However, SILK provides support for defining new methods on Java classes from Scheme and the ability to reflect on Java objects, the features that we have not explored in our current implementation. iScheme's approach differs from most of the existing Scheme and other languages symbiosis in that we aim at building constructs that ease the development of applications that run on mobile devices while exploiting the capabilities specific to mobile platforms.

5.10 Chapter Summary

In this chapter, we have presented iScheme, which will serve as our linguistic experimentation platform for the language constructs and features to ease the development of *reactive* context-aware applications. Central to iScheme, is the language symbiosis between Scheme and Objective-C, which facilitates the development of dynamic mobile applications that exploit context sensors available on the iOS devices. In addition, iScheme provides distribution constructs for peer-to-peer service discovery, remote communication, and handling network failures. These constructs have been built using the syntactic extension mechanism of Scheme. iScheme lays a foundation for the language constructs and features that we present in the next chapters.

Chapter 6

The Flute Language: A Developer's Perspective

Contents

6.1 Introduction

One of the goals of this dissertation is to design and implement a programming language to enable the development of *reactive* context-aware applications. In Chapter 2, we motivated the need for such a language and identified programming language requirements that should be satisfied by that language, namely, *R*.1 *Chained context reactions, R*.2 *Context-dependent interruptions, R*.3 *Context-dependent resumptions, R*.4 *Contextual dispatch, R*.5 *Reactive dispatch, and R*.6 *Reactive scope management.* In Chapter 4, we presented the ICoDE model that aims to define the main

properties and the boundaries of a programming language designed for *reactive* context-aware applications, namely, *predicated procedures, representing of context as reactive values, reactive dispatching, interruptible executions, resumable executions,* and *scoped state changes*. In Chapter 5, we presented iScheme [BVB+12], a mobile programming language laboratory that we developed to facilitate our experiments with new language constructs on a state-of-the-art mobile computing platform. The stage is now set to introduce the *Flute* language [BVDR+12] – our proof-of-concept programming language that adheres to the properties of the ICoDE model. Flute has been implemented as a meta-interpreter in iScheme. It provides language runtime support and language constructs for realising the properties of the ICoDE model. The syntax of the Flute language is essentially an extension of that of Scheme. Before explaining the constructs and features of the Flute language, we introduce a running example that we will use throughout this chapter.

Recall that in Chapter 2, we introduced the *BainomuAppies* in Kampala scenario to motivate the need for the software technology for *reactive* context-aware applications. However, given the large scale and infrastructure demands (buses, minibuses, onboard computers, etc.) of the *BainomuAppies* in Kampala scenario, it is not feasible to implement that scenario within the scope of a Ph.D. dissertation. Instead we consider a variant case study, called the *iFlute platform*. The iFlute platform exhibits characteristics similar to those of the onboard digital platform for the *BainomuAppies* scenario (cf. Section 2.3) only that it runs on a mobile device – specifically the Apple iPad tablet. Like the onboard digital platform, the iFlute platform is enhanced with context-awareness to perpetually switch between applications and behaviours in reaction to context changes. Examples of applications that run on the iFlute platform include: *Kalenda*, which is a context-aware calendar application; *Pulinta*, which is a context-aware printer assistant; and *Tasiki*, which is a context-aware task guide. In Chapter 8, we provide a full description of the iFlute platform, the applications and their implementation using Flute. In this chapter, we consider the *Kalenda* application as a running example to describe the Flute language from a developer's perspective.

6.2 A Running Example: *Kalenda* Application

Kalenda is a context-aware calendar application that runs on the iFlute platform. The *Kalenda* application automatically launches to show the relevant calendar items whenever the user moves within range of his/her workplace.

Calendar items may include user's private appointments (e.g., family events or a doctor appointment) that should be displayed only to the device owner and public items (e.g., workplace meetings or bank holidays) that can be visible to everyone (e.g., co-workers). A *reactive* context-aware calendar application should adapt to automatically hide the private agenda items when the owner is not using the device. For instance, suppose that a user who is browsing through his/her private calendar items temporarily gives the device to a coworker. The calendar application should immediately adapt to show only public items and adapt the display properties (e.g., font size or colour) to match the coworker's preferences. Furthermore, suppose that the coworker gives back the device to the owner, the calendar application should immediately restore the owner's previous calendar items view. Figure 6.1 shows an instance of the calendar application running in the public mode on the Apple iPad tablet.

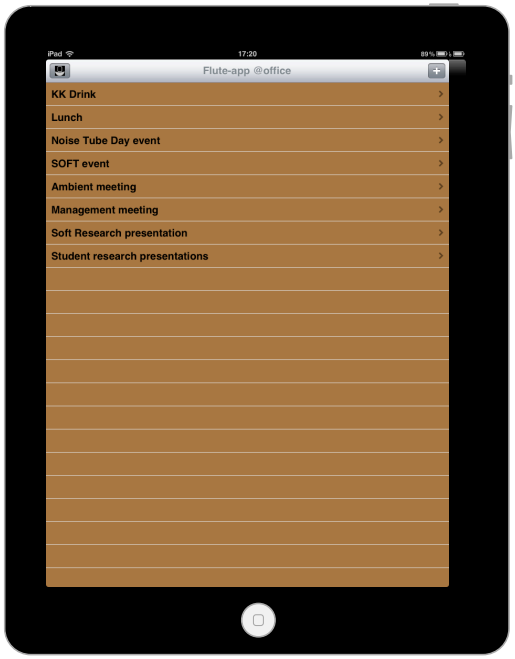

Figure 6.1: An instance of the *Kalenda* application executing in the public mode.

6.3 Building Blocks: Modes and Modals

To incorporate the ICoDE model, Flute introduces two building blocks, namely, *modes* and *modals*.

```
(define <context-source-name>  (ctx-event))

(define <modal-name> (modal (<context-sources>) <shared-variables>))

(mode (<modal-name>)
    <context-predicate>
    <value-expression>)

(mode (<modal-name>)
   <context-predicate>
   (<configuration-options>)
   (lambda (<parameters>)
       <body>))
```

Figure 6.2: An informal description of the Flute syntax for modal, mode, and context source definitions.

Definition 7 ***(Mode)*** *A mode defines a variant of behaviour (context-dependent procedure) or state (context-dependent variable) for a particular context. A mode is constrained to a particular context through a developer specified context predicate.*

Definition 8 ***(Modal)*** *A modal is a group of related modes. In addition, it specifies the state variables that are shared among different modes and it specifies context sources that are used in the context predicates of those modes.*

In Flute, a context-dependent procedure and a context-dependent variable is represented as a mode. Each mode defines a different behaviour or value for a different context. Related modes are grouped together under the same modal. In the context-aware calendar application, for example, there are different *procedure modes*, "private" and "public", for showing the agenda items depending on whether the device user is the owner or not. Such procedure modes can be grouped together under a single modal, "agenda". Flute provides language constructs to create modals and modes. However, the developer does not need to worry about ensuring that the appropriate mode is always executed for the current context of use. The language runtime ensures that the right mode is executed for the right context and that the entire execution of the mode happens under the specified context condition. Moreover, as we will see in the next sections new modes can be dynamically added to a modal as required.

Having introduced the building blocks of the Flute language, we will now discuss its support for the ICoDE model (cf. Chapter 4) by means of illustrative examples. We will use the *Kalenda* application that was introduced above as the running example throughout this chapter. Figure 6.2 shows the informal description of the Flute syntax for mode, modal, and context source definitions.

6.4 Variables Modals and Modes

In Flute, a variable modal can have one or more values that correspond to different execution contexts. Therefore, a variable access yields a different value depending on the context in which it is accessed. This is unlike variables in conventional programming languages where a variable access always yields the same value. For instance, when developing our context-aware calendar application, we require a variable that contains a different colour value depending on the device user (i.e., a grey colour when the device user is the owner and a brown colour when the device user is not the owner). In Flute, this behaviour can be implemented as follows.

Listing 6.1: Defining variable modes

```
1  (define current-user (ctx-event))
2
3  (define bg-colour (modal (current-user)))
4
5  (mode (bg-colour)
6    (not-owner? current-user) ;a context predicate
7    brown-colour)
8
9  (mode (bg-colour)
10   (owner? current-user)
11   grey-colour)
```

Listing 6.1 creates `bg-colour` as a variable modal that has two colour modes. A modal is created using the special form `modal` while a mode is created using the special form `mode`. The modal definition specifies a context source upon which context predicates operate. A context source is created using the special form `ctx-event`. In the above example, `current-user` is populated with a value that indicates the current user of the device. The details of initialising context sources with values from physical sensors (such as GPS) are discussed in Section 6.7.2. For now it suffices to know that the `current-user` context source is automatically kept up-to-date by the

language runtime such that its value at any given moment indicates the current device user. Each mode definition specifies the modal it belongs to, a context predicate and a value for the mode. Figure 6.3 depicts the representation of the two modes belonging to the `bg-colour` modal. Note that it is possible to add new modes to a variable modal at runtime. This has an advantage that developers can add unanticipated variable modes on demand. The evaluation semantics of creating variable modals are discussed in Section 7.5.3.

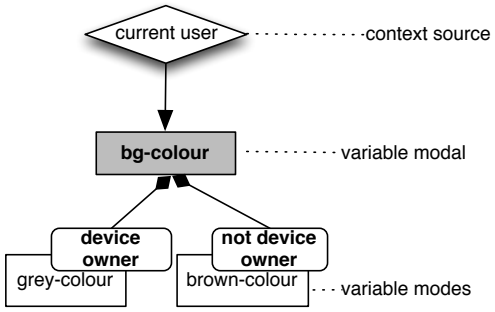

Figure 6.3: The `bg-colour` variable modal that has a different colour depending on the current context of use.

6.4.1 Variable Modal Access Semantics

Just like a regular variable, the value of a variable modal can be accessed through its name. The difference, however, is that accessing a variable modal can yield a different value depending on the current context of use. For instance, in the above example, the variable modal `bg-colour` yields `brown-colour` or `grey-colour` depending on the current device user. The REPL transcript below illustrates the semantics of accessing the variable modal `bg-colour`. The input expression is prefixed with the prompt > while the result of evaluating the expression is prefixed with a ===>.

```
1 ;suppose the current user is the device owner
2 > bg-colour
3 ===> grey-colour
4
5 ;suppose the current user is not the device owner
6 >  bg-colour
7 ===> brown-colour
```

Flute employs a contextual dispatching mechanism to select the right value for a variable modal. In other words, the value for a variable modal is

that of a mode whose context predicate evaluates to *true*. In case there are multiple values (i.e., if there is more than one context predicate that evaluate to *true*) or there is no value found (i.e., if there is no context predicate that evaluates *true*), an exception is thrown. The evaluation semantics of contextual dispatch for variable modals are discussed in Section 7.5.5.

6.4.2 Assignment Semantics for Variable Modals

An assignment to a variable modal only affects the value of the variable mode whose context predicate evaluates to *true*. Before performing a state change, the correct mode is looked up depending on the current context of use. The following transcript illustrates a mutation of the `bg-colour` introduced in Listing 6.1.

```
1  ;suppose the current user is the device owner
2  >  (set! bg-colour blue-colour)
3
4  ;now accessing bg-colour with the user still the device owner
5  > bg-colour
6  ===> blue-colour
7
8  ;suppose the current user is not the device owner
9  ;the value of the not device owner mode is not affected
10 > bg-colour
11 ===> brown-colour
```

As we can see from the above code example, performing an assignment operation on the `bg-colour` variable modal when the current device user is the owner, only affects the value of that mode.

6.5 Procedure Modals and their Modes

Context-dependent behavioural variations in Flute are expressed through *procedure modes*. Related procedure modes are grouped together under one *procedure modal*. Flute provides constructs to enable creation of procedure modals and procedure modes. We will illustrate procedure modals and their modes using our running example of the *Kalenda* application. Remember that the *Kalenda* application consists of two context-dependent behavioural modes: (i) private agenda mode, and (ii) public agenda mode. The private agenda mode is executed when the owner is using the device, whereas the

public agenda mode is executed when another user is using the device.[1]

6.5.1 Creating a Procedure Modal

Flute supports the creation of an entity that groups together related procedure modes under one identity, a *procedure modal*. Similar to the variable modals (cf. Section 6.4), a procedure modal is created using the special form `modal`. In addition, a procedure modal may specify variables that will be shared by all procedure modes that belong to this modal. The syntax of the special form for creating a procedure modal is as follows.

```
(define <modal-name> (modal (<context-sources>)
   <optional-shared-variables>))
```

The variable `<modal-name>` is bound to the resulting modal. The special form `modal` takes two arguments. The first argument is a list of context sources that are used in the context predicates of the procedure modes that belong to that modal (cf. Section 6.5.2). The second argument includes the optional definition of variables that are shared by the modal's modes. The following excerpt illustrates how an agenda modal of the *Kalenda* application can be defined.

Listing 6.2: Defining variables that are shared among the procedure modes belonging to the same modal

```
1
2    (define agenda (modal (current-user)
3      (define date-range          2)
4      (define display-scale       4)))
```

The `agenda` modal is used to group together procedure modes for presenting different agenda items depending on the current device user. `current-user` is the context source that will be used in the context predicates of the procedure modes to check whether the device is being used by its owner or not. Variables `date-range` and `display-scale` are the variables that are shared by all modes belonging to that modal. A modal does

[1] In this example we use the device orientation sensor to simulate the change of the current device user. In the near future, we can imagine the use of NFC-enabled wristwatches to detect the identity of a user.

not specify which modes belong to it. Instead, modes are the ones that specify which modal they belong to. This enables addition of modes to a modal at runtime whenever required. Note that modals in the spirit of Scheme are first-class entities. In the next section, we discuss the definition of procedure modes.

6.5.2 Creating Procedure Modes

Flute supports **Property #1: Predicated Procedures** of the ICoDE model (cf. Section 4.4) through procedure modes. The general form for creating a procedure mode is as follows. The `mode` special form takes as

```
(mode (<modal-name>)
   <context-predicate>
   (<configuration-options>)
   (lambda (<parameters>)
       <body>))
```

argument, a modal, a context predicate, a list of interruption, resumption, and state scoping strategies `<configuration-options>`, and a procedure that is created using the `lambda` special form. The context predicate `<context-predicate>` constrains the execution of the mode to a particular context of use. The code excerpt below illustrates the definition of different modes for the agenda modal. Each mode provides a different behaviour for presenting the agenda items depending on the current device user.

Listing 6.3: Defining procedure modes

```
1  (define agenda (modal (current-user)
2    (define date-range          2)
3    (define display-scale       4)))
4
5  (define config
6    (create-config suspend resume isolated))
7
8  (define show-private-agenda
9    (mode (agenda)
10     (owner? current-user) ;context predicate
11     (config)              ;configuration options
12     (lambda ()
13       (show "private agendas")
```

```
14          (set! display-scale 8)
15          (show bg-colour)
16          (scale display-scale)
17          ...
18          (show calendars))))
19
20  (define show-public-agenda
21    (mode (agenda)
22      (not-owner? current-user)
23      (default-config)
24      (lambda ()
25        (show "public agendas")
26        (show bg-colour)
27        ...
28        (show calendars))))
```

Listing 6.3 creates two procedure modes `show-private-agenda` and `show-public-agenda` using the special form `mode`.[2] Both modes belong to the `agenda` modal. The `agenda` modal definition specifies the `current-user` variable as the context source that will be used in the context predicates for its procedure modes. In addition, it defines the variables `date-range` (on Line 2) and `display-scale` (on Line 3) that are shared by the modes belonging to the `agenda` modal. The `date-range` value specifies the date range of calendar items to display. The `display-scale` value specifies the scale of the font size for the calendar display. As with variable modes, new procedure modes can be dynamically added to a modal.

Each procedure mode is associated with a context predicate that should be satisfied throughout the execution of that mode. In the above example, the context predicates for the `show-private-agenda` and `show-public-agenda` modes are `owner?` and `not-owner?`, respectively. As already mentioned above these context predicates operate on the `current-user` context source. Figure 6.4 depicts the two modes of the `agenda` modal. The context predicates specified in each mode play two roles. First, they are used by the language runtime to dispatch a call to the right mode to execute for the current context of use. Second, they are used by the language runtime to ensure that the execution of the mode is constrained to the correct context by perpetually evaluating the context predicate at every evaluation step of the mode. In Section 6.5.3, we further discuss the

[2]For brevity, the above implementation of the *Kalenda* application does not include the graphical GUI concerns of the calendar application, but is restricted to textual display. The full implementation of the context-aware calendar application is further discussed in the validation chapter (cf. Section 8.4).

dispatching process for procedure modes.

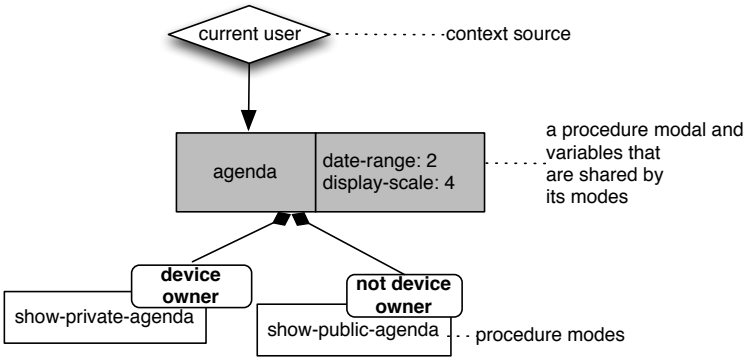

Figure 6.4: An illustration of the `agenda` modal and its modes.

In addition, a procedure mode definition includes configuration options that specify a strategy for interruption (i.e., `suspend` or `abort`), a strategy for resumption (i.e., `resume` or `restart`) and a strategy for scoping state changes (i.e., `immediate`, `deferred` or `isolated`). The developer may use the default configuration `default-config` or can define own configuration options using the `create-config` abstraction. The `default-config` specifies the configuration options as `(:p-false suspend :p-true restart :state-changes immediate)`, which implies that when the context predicate is false the execution is suspended, when the context predicate becomes satisfied again the execution is restarted and any state changes are immediately visible. In the above example, the private agenda mode specifies the configuration as `config` that is defined on Line 5 while the public agenda mode uses the default configuration. In Section 6.5.4 we further discuss the interruptible context-dependent execution of procedure modes.

6.5.3 Reactive Dispatching for Modes

Flute supports **Property #3: Reactive Dispatching** of the ICoDE model (cf. Section 4.6) by incorporating a reactive dispatching mechanism for modes. In Flute, the execution of modes is initiated by invoking a modal. When a modal is invoked, a dispatching mechanism selects the right mode to execute for the current context. For instance, the invocation of the `agenda` modal `i.e., (agenda)`, may result in either the `show-private-agenda` mode or the `show-public-agenda` mode being executed. Procedure modes are not directly invoked by the developer but

by the dispatching mechanism. Because of this, all modes belonging to the same modal are required have the same number of parameters.

The dispatching process starts by evaluating all context predicates that are associated with the modes that belong to the invoked modal. The mode whose context predicate evaluates to *true* is scheduled for execution. However, unlike traditional dispatching mechanisms in conventional languages, the dispatching process in Flute does not happen just once. Since context changes occur continuously, it is possible that some context predicates that could not be satisfied, become satisfied later and thus require their associated modes to be executed. In Flute, the dispatcher is implicitly registered to the invoked modal's context sources and the dispatching process is triggered anew whenever context sources receive new values. This means that modes that were not previously selected for execution may be selected later. So even when there is no applicable mode, the Flute dispatcher does not throw *a mode-not-found exception* because *the mode may be applicable later* when relevant context changes are observed. In Section 7.5.6 we discuss the technical details for the reactive dispatching mechanism in the Flute interpreter.

Note that it is possible to directly bind a procedure mode (created using the `mode` special form) to a name. This enables invoking such a mode directly. In that case, the context predicate associated with the mode is only used to ensure that execution of the mode happens in the correct context. Also, allowing modes to be invoked directly makes it possible to define recursive procedure modes. In the next section, we discuss the interruptible execution semantics for procedure modes.

6.5.4 Interruptible Execution of Modes

Flute supports **Property #4: Interruptible Executions** of the ICoDE model (cf. Section 4.7) through interruptible context-dependent execution of procedure modes. Once a mode has been scheduled for execution, its associated context predicate must be satisfied throughout the mode's execution. The Flute runtime *implicitly* re-evaluates the context predicate throughout the mode execution. In the context predicate happens to evaluate to false, the execution is promptly interrupted. The kind of interruption depends on a developer specified interruption strategy. Flute supports the interruptions strategies `suspend` and `abort` of the ICoDE model (cf. Section 4.7).

For instance, in the *Kalenda* application, both the `show-private-agenda` and `show-public-agenda` modes are specified with the `suspend` as their interruption strategy. Suppose that the `agenda` modal is initially invoked when the device is being used by its owner. Hence the `owner?` predicate is satisfied. As a result, the

`show-private-agenda` mode will be selected and its execution started to show all the agenda items including private appointments and workplace meetings. Suppose that the device owner then gives the device to another user while the `show-private-agenda` mode is executing. As a consequence, there will be a context switch and the context predicate `owner?` will no longer be satisfied. The execution of the `show-private-agenda` mode cannot be allowed to continue. As its configuration options `config` specify `suspend` as the interruption strategy, the mode's execution is suspended. On the other hand, the context predicate `not-owner?` will be satisfied as soon as the device changes hands. Therefore, the `show-public-agenda` will take over and its execution will be started to show only the public agenda items. In Section 7.5.6 we discuss the details of how the evaluation of procedure modes works in the Flute interpreter.

6.5.5 Event-driven Resumption of Suspended Executions

Flute supports **Property #5: Resumable Executions** of the ICoDE model (cf. Section 4.8) through event-driven resumption of suspended executions. The language runtime ensures that suspended executions are resumed when their associated context predicates later become satisfied again. As context changes occur, context sources receive new values and as a result context predicates that operate on those context sources may become satisfied again. Suspended executions whose context predicates become satisfied again are scheduled for resumption. The kind of resumption depends on a developer specified resumption strategy. Flute supports the resumptions strategies `resume` and `restart` of the ICoDE model (cf. Section 4.8).

In the *Kalenda* application, suppose that while the `show-public-agenda` mode is executing, the device is given back to the owner. Then the execution of the public agenda mode will be interrupted promptly. Conversely, the execution of the `show-private-agenda` mode will be resumed from where it left off since the resumption strategy in the configuration options is `resume`. For instance, if the user was scrolling an agenda items list before the interruption, the application will be resumed at the same position where the user was. Resumption of suspended executions is triggered by the occurrence of relevant context changes. In this example, the language runtime is implicitly registered to the context source `current-user` and is notified when a new value is received (we discuss the representation of context sources as *reactive values* in Section 6.7.2). Subsequently, the `owner?` context predicate is re-evaluated and the

previously suspended `show-private-agenda` mode's execution will be resumed if the predicate evaluates to *true*.

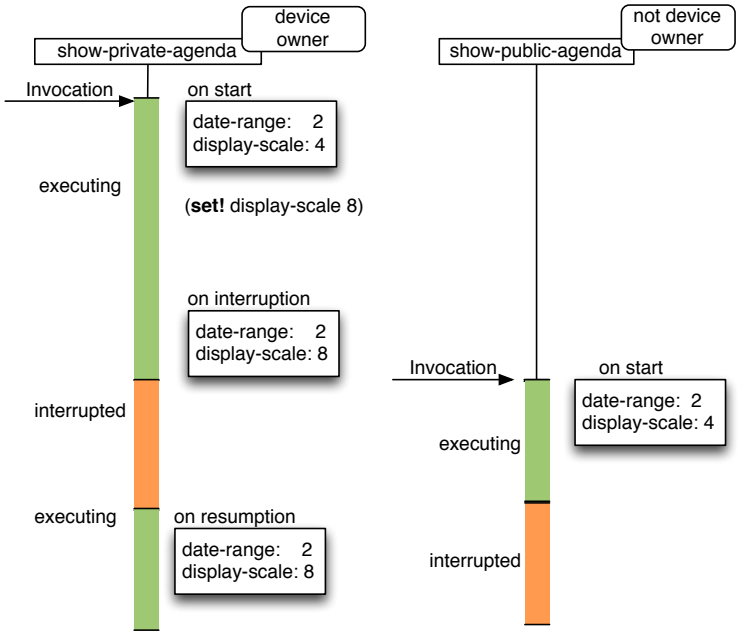

Figure 6.5: An illustration of the `isolated` state change scoping strategy in action.

6.5.6 Scoped State Changes

Flute supports **Property #6: Scoped State Changes** of the ICoDE model (cf. Section 4.9) through configuration options that enable the developer to specify a strategy for controlling the visibility of the state changes made by a mode to the variables that are shared among procedure modes. Flute supports three state scoping strategies `immediate`, `deferred` and `isolated` of the ICoDE model.

In the *Kalenda* application (cf. Listing 6.3), the `show-private-agenda` mode is specified with the `isolated` strategy, which implies that all state changes remain local to that mode. For instance, the `show-private-agenda` mode modifies the `display-scale` variable to increase display scale (i.e., `(set! display-scale 8)` on Line 14). Such a state change will remain local to the `show-private-agenda` mode and will not be visible to the `show-public-agenda` mode. Figure 6.5 illustrates the state change management with the `isolated` strategy as the execution switches back and forth between the `show-private-agenda` and

`show-public-agenda` modes. Suppose that `show-private-agenda`'s execution is interrupted after changing the value of the `display-scale` variable to `8`. As the `show-private-agenda` mode is specified with the `isolated` strategy, when the `show-public-agenda` mode takes over, the value of the `display-scale` variable will still be `4`. However, if the `show-private-agenda` later resumes executing, the value of the `display-scale` will be `8` (i.e., the one it was changed to before the interruption).

With the `isolated` strategy, the Flute runtime keeps a *local copy* when a variable is accessed for the first time, and any subsequent changes are made to that local copy. Other state change scoping strategies are `immediate` and `deferred` as discussed in Chapter 4 Section 4.9. With the `deferred` strategy, the Flute runtime keeps a local copy as in the `isolated` strategy, with an additional validation step before committing the changes when the mode completes executing. The validation step involves comparing the values of variables when they were first read and its current value. If the validation step succeeds, then the mode state changes are committed. Otherwise, the state changes are discarded. We further discuss how state change scoping strategies are realised in the Flute interpreter in Section 7.5.7.

6.5.7 Nested Procedure Modes

Flute supports nested mode definitions (i.e., it is possible to define a mode within another mode). Recall from Section 4.4 that the ICoDE model requires the context predicate associated with an enclosing predicated procedure be satisfied throughout the execution of its nested predicated procedures. In the ICoDE model this mechanism is classified as *lexical propagation of context predicates*. The ICoDE model also requires that the context predicate associated with a predicated procedure be satisfied during the execution of the predicated procedures that are invoked from that predicated procedure's body. In the ICoDE model this mechanism is classified as *dynamic propagation of context predicates*. As Flute adheres to the ICoDE model, it supports both lexical and dynamic propagation of context predicates. In the remainder of this section we illustrate the definition of nested procedure modes and the propagation of context predicates among the modes.

Defining nested procedure modes. In Flute, a nested mode "inherits" the context predicate of its enclosing mode. Therefore, the execution of a nested mode must satisfy both its associated context predicate and that of its enclosing mode. In order to illustrate the lexical propagation of context

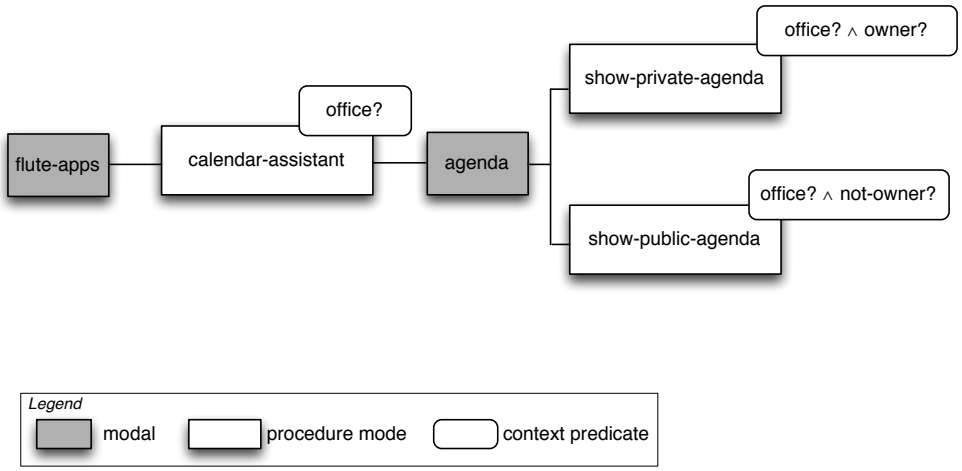

Figure 6.6: Lexical propagation of context predicates for nested procedure modes.

predicates in Flute, we will extend our running example of the *Kalenda* application. Recall from Section 6.2 that the *Kalenda* application is among a suite of applications that are deployed in the iFlute platform. Up to now we have focussed on the definition of its two context-dependent behavioural modes for showing the agenda items. The *Kalenda* application as a whole is also defined as a mode that belongs the modal that groups together all the applications of the iFlute platform, as shown below.

Listing 6.4: Nested mode definitions.

```
(define flute-apps (modal (location)))

(define calendar-assistant
  (mode (flute-apps)
    (office? location)
    (config)
    (lambda ()
      (define show-private-agenda
        (mode (agenda)
          (owner? current-user)
          (config)
          (lambda ()
            ...)))
      (define show-public-agenda
        (mode (agenda)
          (not-owner? current-user)
```

```
17          (default-config)
18          (lambda ()
19            ...)))))))
```

Listing 6.4 shows the definition of the `show-private-agenda` and `share-agenda` modes as nested modes of the `calendar-assistant` mode. The `calendar-assistant` represents the entire calendar application and it is associated with the context predicate `office?` that ensures that the application should only be launched when the user is in his/her office. The `calendar-assistant` mode belongs to the `flute-apps` modal that groups together context-aware applications that run in the iFlute platform. As `show-private-agenda` and `show-public-agenda` are nested modes of the `calendar-assistant` mode, the `office?` context predicate should also be satisfied during the execution of those nested modes (in addition to their own context predicates). Figure 6.6 illustrates the lexical propagation context predicates among the procedure modes of the *Kalenda* application.

6.5.8 Demarcating Uninterruptible Regions

As discussed in Section 4.7, it is desirable for a programming language that adheres to the ICoDE to enable the developer demarcate certain regions of a procedure's body that may need to be run without interruption (e.g., *all-or-nothing* IO actions). To this end, Flute provides the language construct `(continuous <`*expressions*`>)` to demarcate certain parts of a procedure body as *uninterruptible* regions. The language runtime ensures that the execution of the `<expressions>` are evaluated without interruption. In Section 7.5.8 we discuss the evaluation of the `continuous` special form in the Flute interpreter.

6.6 Scoping Semantics

Flute adheres to the *lexical scoping* semantics of the Scheme language in which it is embedded. However, the introduction of the variables that are specified as part of the modal definition, requires a slight variation of variable lookup semantics. Below we illustrate the scoping issues that arise.

Listing 6.5: Variable and procedure lookup semantics

```
1 ;modal definition
2 (define x 2)
3 (define m (modal (x)
```

```
4     ;modal-specific shared variable
5     (define y 4)))
6
7   (define x 20)
8   (define y 40)
9
10  ;mode definition
11  (mode (m)
12    ;which x is it?
13    ;which y is it?
14    (smaller? x y)
15     ...
16    (lambda ()
17      (define x 200)
18      ;which x is it?
19      ;which y is it?
20      (show (+ x y))))
```

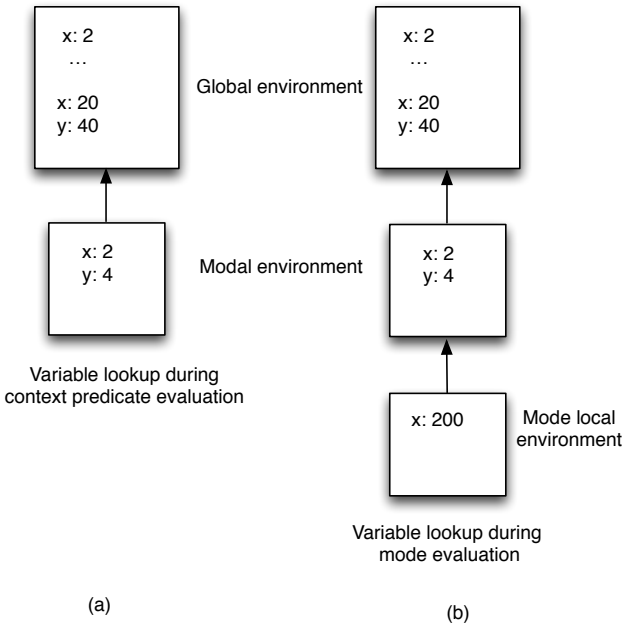

Figure 6.7: An illustration of the environment structure for the variable look up during the evaluation of context predicates and modes.

In Listing 6.5 above, we define a modal `m` that captures the variable `x` as a context source (i.e., at Line 3). In addition, the modal defines the variable `y` that is shared among all modes belonging to that modal. Lines 7 and 8

define new global variables. The need for new scoping semantics becomes apparent in the lookup of the variables used in the context predicate expression `(smaller? x y)` and in the body of the mode. Figures 6.7 *(a)* and *(b)* depict the resulting environment structure for the context predicate and the evaluation of the mode. The modal environment includes the variables that are defined in the modal (i.e., at Line 5) and the variables that are captured by the modal (i.e., at Line 3).

Variable lookup during context predicate evaluation. Variable lookup during the evaluation of a context predicate is as follows:

1. To evaluate a context predicate, the lookup of variables starts from the modal environment, which includes variables that are explicitly captured and those that are defined by the modal (cf. Figure 6.7 *(a)*).

2. If the variable is not found in the modal environment, the lookup proceeds to the enclosing environment of the mode.

To illustrate variable lookup during a context predicate evaluation, consider the code example in Listing 6.5. In that example, the variables `x` and `y`, which are used in the context predicate expression `(smaller? x y)`, will resolve to `2` and `4` during the evaluation of that context predicate expression.

Variable lookup during mode evaluation Variable lookup during evaluation of a mode is as follows:

1. To evaluate the body of a mode, the lookup of variables starts from the local environment of the mode.

2. If the variable is not found in the mode's local environment, the lookup proceeds to the environment of the modal to which it belongs.

3. If the variable is not found in the modal environment, the lookup proceeds to the enclosing environment of the mode.

Therefore, in the above example the variables `x` and `y` that are used in the procedure mode will resolve to `200` and `4`, respectively. Notice that the value of `y` resolves to that of the modal environment. These scoping semantics are particularly essential for the evaluation of context predicates and modes that are added at runtime to existing modals.

6.7 Representing Context as Reactive Values

Flute supports **Property #2: Representing Context as Reactive Values** of the ICoDE model (cf. Section 4.5) through abstractions for *reactive values*. Flute's support for reactivity is inspired by existing work on functional reactive programming (FRP) [CK06, MGB+09] (cf. Section 3.3). In this section we first discuss Flute's support for reactive values and in Section 6.7.2 we show how to represent context sources as reactive values.

6.7.1 Reactive Values in Flute

A reactive value is like a regular value in a programming language, except that when its value changes, any computation that uses its value is automatically recomputed. Flute provides the `ctx-event` construct for creating reactive values. Listing 6.6 below depicts a reactive program in Flute.

Listing 6.6: Defining reactive values in Flute.

```
1 (define x (ctx-event 1))
2 (define y (+ x 1))
3 (define z (< x y))
```

In Flute, a reactive value is created using the `ctx-event` construct, which takes an optional initial value as its argument. In this example, the variable `x` is bound to the resulting reactive value whose initial value is `1`. Additionally, when a procedure is applied to a reactive value, the result is also a reactive value. As such, the variable `y` is bound to a reactive value whose value at any given moment is `x+1`, while the variable `z` is bound to a reactive value whose value is a Boolean that indicates whether or not the value of `x` is less than that of `y`. As `x` is a reactive value whenever its value changes the value of `y` is automatically updated and so the value of `z` is also automatically updated. Flute provides the `update-value!` construct for updating the value of a reactive value. For instance, the value of `x` can be updated by evaluating the expression `(update-value! x 2)`. The `update-value!` construct is particularly used to acquire (non-reactive) low-level data from external sensors (cf. Section 6.7.2).

Automatic and consistent propagation of changes. The Flute language runtime ensures that when a value of a reactive value changes, such a change is *automatically and consistently* propagated among all its dependents. It does so by automatically tracking dependencies among dependent reactive values. Figure 6.8 depicts the dependency graph for the reactive

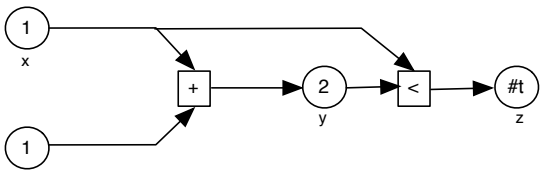

Figure 6.8: A dependency graph among reactive values.

program in Listing 6.6. The propagation mechanism employs a push-based evaluation model. That is, propagation of changes among dependent reactive values is driven by availability of new values (*data-driven*) rather than the *demand*. Whenever a *reactive value* receives a new value, all reactive values that depend on it are scheduled for updating. When scheduling reactive values for updating, it is important that such updates are *consistently* propagated. Otherwise, update inconsistencies can occur – a phenomena known as glitches [CK06] in the reactive programming literature. Glitches occur when reactive values are updated in a wrong order. This can result in fresh values being combined with stale values, leading to a incorrect program state. In order to illustrate the glitch problem, let us go back to the reactive program in Listing 6.6. Figure 6.9 depicts a possible propagation of updates among the reactive values `x`, `y`, and `z`.

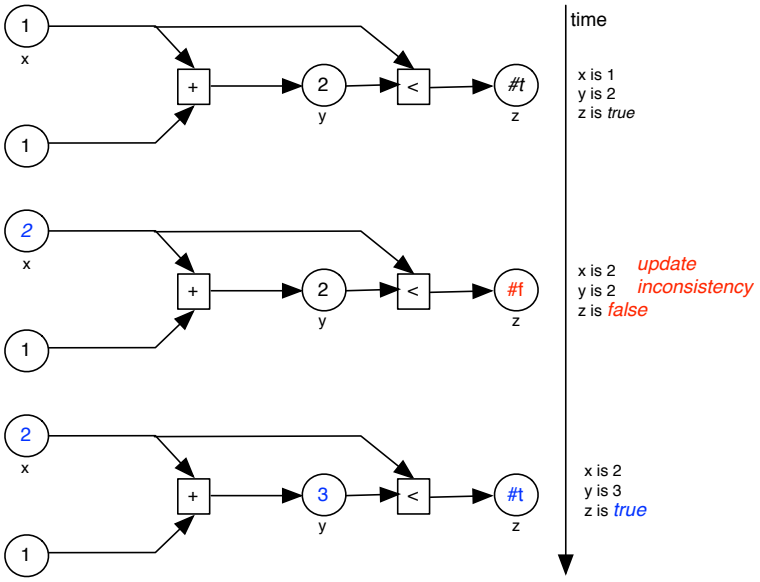

Figure 6.9: The need for consistent propagation of changes.

As `x` is less than `x+1`, the value of the variable `z` is expected to always

be *true* (i.e., `#t`). Initially when the value of `x` is `1`, the value of `y` is `2` and that of `z` is `#t`. If the value of `x` later changes to say `2`, the value of `y` is expected to change to `3` while the value of `z` is expected to be `#t`. However, in a naive reactive implementation, changing the value of `x` to `2` may cause the expression `(< x y)` to be re-evaluated before the expression `(+ x 1)`. Thus the value of `z` will momentarily be `#f`, which is incorrect (i.e., an update inconsistency). Eventually, the expression `(+ x 1)` will be recomputed to give a new value to `y`. Therefore, the value of `z` will be recomputed again to reflect the correct value `#t`. In the reactive programming literature, such a momentary view of inconsistent state is known as a *glitch* [CK06]. Flute eliminates such update inconsistencies, by arranging expressions in a topologically sorted graph, a technique also used in other FRP languages like FrTime [CK06], and Flapjax [MGB+09].

Implicit lifting. When native procedures such as `+, <` and user defined procedures are applied to a reactive value, they are *implicitly lifted* to operate on reactive values. The return value of applying a procedure to a reactive value is also a reactive value whose value at any moment is the result of applying that procedure to that reactive value. Internally, the Flute language runtime transparently establishes a dependency between the two the reactive values. Taking the example in Listing 6.6 again, evaluating the expression `(+ x 1)` implies that the `+` procedure is implicitly lifted to operate on `x` and its return value is also a reactive value whose value is recomputed every time the value of `x` changes. Implicit lifting enables combining procedures with reactive values.

6.7.2 Defining Context Sources in Flute

Now that we have discussed Flute's support for reactive values, let us show they are used for representing context sources.

Listing 6.7: Defining sources

```
1 ;defining  a context source
2 (define gps-coordinates (ctx-event))
3
4 ;definition of the location context source
5  (define location
6   (gps->location gps-coordinates))
7
8 ;obtaining GPS coordinates
9 (CURRENT-LOCATION
```

```
10  (lambda (latitude longitude)
11    (update-value! gps-coordinates
12                   (cons latitude longitude))))
```

Listing 6.7 shows the definition of the context source `gps-coordinates` as a *reactive value* using the `ctx-event` construct. The `gps->location` procedure transforms raw GPS coordinates into a high-level contextual value such as `'office` or `'home`. As `gps-coordinates` is a reactive value, the result of the expression `(gps->location gps-coordinates)` is also a reactive value (cf. Section 6.7.1). This means that the Flute language runtime establishes a dependency between `gps-coordinates` and `location`. The GPS coordinates are obtained using the `CURRENT-LOCATION` construct that is provided by iScheme. As explained in Section 5.5, this construct takes a procedure as its argument and registers it as an event-handler that is invoked whenever GPS sensors have new latitude and longitude values. Non-reactive raw GPS coordinates are added to `gps-coordinates` using the `update-value!` construct. As `location` depends on `gps-coordinates`, whenever new GPS coordinates are received, the value of `location` is automatically updated by applying the `gps->location` procedure to the value of `gps-coordinates`. Such automatic propagation of changes ensures that context sources that depend on each other are always kept up-to-date.

6.8 Programming Language Requirements Revisited

The preceding sections have presented the features of the Flute language. We introduced its language constructs that facilitate the development of *reactive* context-aware applications. The Flute language has been designed to adhere to the ICoDE model that we discussed in Chapter 4. Flute is the first language to support interruptible context-dependent executions. By adhering to the ICoDE model, Flute satisfies the language requirements that we put forward in Chapter 2. In this section, we review each requirement and discuss how it is addressed by Flute.

Chained Context Reactions (R #1) Flute provides a *reactive value* abstraction that is used to represent context sources. By representing context sources as reactive values, Flute enables developing programs that operate on context sources without explicitly managing callbacks. To this end, Flute provides the `ctx-event` construct for representing

context sources as reactive values. Reactive values employ a push-driven evaluation model for automatic propagation of context changes. As discussed in Section 6.7.2, representing context sources as reactive values maps well onto the event-driven resumption of suspended executions.

Context-dependent Interruptions (R #2) Flute provides the *mode* abstraction, which is a predicated procedure that is associated with a context predicate that specifies the context condition under which the procedure is constrained to run. As discussed in Section 6.5.2, the context predicate associated with each mode is implicitly checked throughout the execution of the procedure body. This ensures that the entire procedure execution is constrained to the developer prescribed context predicate. The execution of a procedure mode in Flute can be interrupted at any moment if its associated context predicate is no longer satisfied. As discussed in Section 6.5.4, Flute provides two interruption strategies, `suspend` and `abort`, that a developer can choose from to specify a suitable interruption mechanism for a mode.

Context-dependent Resumptions (R #3) As discussed in Section 6.5.5, Flute provides two resumption strategies – `resume` and `restart` – that a developer can choose from to specify a suitable resumption mechanism (i.e., what to do when the associated context predicate becomes satisfied again) for a mode.

Contextual Dispatch (R #4) Flute provides the *modal* abstraction that is used to group together related procedure and variable. A modal can be bound to a name, is a first-class entity and can be referred to in other parts of the program. As discussed in Section 6.5.3, it is possible to initiate the execution of modes that belong to the same modal by simply invoking the modal. Flute employs a contextual dispatching mechanism that selects the right mode to execute for the current context based on the context predicate that evaluates to *true*. Additionally, the modal abstraction enables the developer to add new modes at runtime to an existing modal without modifications of the existing modes.

Reactive Dispatch (R #5) Flute supports a new dispatching mechanism that we designate as *reactive dispatching*. The reactive dispatching of procedure modes as discussed in Section 6.5.3 continuously takes into account of any new context changes, even when those context changes happen after the first dispatching phase has completed. As such, previously unsatisfied predicates may become satisfied and their associated

procedure modes will be selected for execution. Additionally, new procedure modes that are added (after the previous dispatching process) are also considered whenever the dispatching process is repeated.

Reactive Scope Management (R #6) As discussed in Section 6.5.6, Flute supports three state scoping strategies that enable the developer to scope the visibility of state changes among executions. These are: `immediate`, `deferred`, and `isolated`. At the definition of a mode, the developer can specify how to control the visibility of state changes that are made during the execution of the mode, by specifying an appropriate state scoping strategy.

Language Construct	**Description**	**Satisfied Requirement**
`ctx-event`	For representing a context source as a reactive value	*R*.1
`mode`	For creating predicated context-dependent procedures or variables	*R*.2 and *R*.4
`modal`	For creating a group of related context-dependent procedures or variables	*R*.4 and *R*.5
`suspend` `abort` `resume` `restart`	Configuration options to specify interruption or resumption	*R*.2 and *R*.3
`deferred` `immediate` `isolated`	State scoping strategies to specify how to to control the visibility of state changes	*R*.6

Table 6.1: Flute language constructs.

6.9 Chapter Summary

In this chapter, we have presented the Flute language and discussed its language support for developing *reactive* context-aware applications. Table 6.1 summarises the language constructs provide Flute that enables Flute satisfy the language requirements that we put forward in Chapter 2. Throughout this chapter we have demonstrated the language constructs using a context-aware calendar application. Flute satisfies the requirements for *reactive* context-aware applications by incorporating the ICoDE model. To recapitulate, Flute provides the following features.

- Flute supports *variable modes* that facilitate the creation of variables whose value depends on the context of use.
- It supports *procedure modes* that enable developers to express predicated procedures with a single context predicate. The context predicate is implicitly checked throughout the mode's execution.
- It supports the *modal* abstraction that enables developers to group together related modes and specify variables that are shared among procedure modes that belong to the same modal. The design of modals facilitates adding new modes at runtime.
- Flute features the *reactive dispatching* mechanism that continuously takes into account of any new context changes to select new applicable modes to execute for the current context. The applicable mode is the one whose context predicate evaluates to *true*.
- Flute features interruptible executions. It provides interruption strategies (*suspend* and *abort*) that enable the developer to specify what to do with the execution when the associated context predicate is no longer satisfied.
- Flute features resumable executions. It provides resumption strategies (*resume* and *restart*) that enable the developer to specify what to do with the suspended execution when its associated context predicate later becomes satisfied again.
- It provides a number of state scoping strategies (*immediate, deferred*, and *isolated*) that enable the developer to control the visibility of state changes to the variables that are shared among procedure modes.
- Flute provides the `ctx-event` construct for representing context sources as reactive values.

Chapter 7

An Executable Semantics for Flute

Contents

7.1 Introduction

Chapter 6 presented the language constructs and features of Flute from a developer's perspective. In this chapter we turn our focus to the semantics of the Flute programming language from an implementation point of view. We describe exact semantics of Flute through a meta-interpreter. The goal of this semantics is to define a working prototype of the Flute language that we can tryout on a mobile platform equipped with sensors. To this end, we have constructed a prototype of Flute as a meta-interpreter conceived on top of iScheme. As presented in Chapter 5, iScheme is a language experimentation laboratory that we developed to enable the prototyping of language constructs and features. Central to iScheme, is the language symbiosis between Scheme and Objective-C. On the one hand, Scheme provides

a suitable infrastructure to create new languages, thanks to its minimalistic syntax and rich features. On the other hand, Objective-C provides rich APIs that enable access to context sensors such as GPS and accelerometer that are available on the iOS devices. Therefore, iScheme provides the benefits of both worlds and thus enables prototyping the Flute language and testing it on a state-of-the-art mobile device.

This chapter starts with a discussion on the semantics requirements in Section 7.2 and the implementation choices in Section 7.3. We then give a recap of the Flute language syntax in Section 7.4. In Section 7.5, we present the different components of the Flute interpreter starting with its main evaluator.

7.2 Executable Semantics Requirements

The requirements of the Flute's executable semantics are derived from the language constructs and features presented in Chapter 6. The executable semantics for Flute should support:

- the definition of modals for context-dependent variables and context-dependent procedures.
- the definition of modes for context-dependent variables and context-dependent procedures.
- reactive dispatching for procedure modes.
- interruptible and resumable execution of procedure modes. It should provide interruption (*suspend* and *abort*) and resumption (*resume* and *restart*) strategies
- state scoping strategies (*immediate*, *isolated*, and *deferred*).
- the definition of context events as reactive values.

7.3 Executable Semantics Choices

When defining an executable semantics of a language on top of an existing language like Scheme, there are various options one can choose from. Here we discuss two choices:

A language extension. One choice is to construct an executable semantics as an extension to an existing programming language through facilities

such as *macros*. The desired semantics of the Flute language makes this choice unsuitable. Flute requires non-trivial features (e.g., interrupting the evaluation of *every* expression) that are difficult to implement correctly using language extension facilities. This difficulty can be alleviated if the existing language supports advanced *runtime reflective APIs* that provide some meta-level "hooks" (e.g., *eval* and *apply*) to intercept the evaluation and procedure application.

An interpreter or (a compiler). Another choice is to build an executable semantics as an interpreter or (a compiler) from the ground up to create a new language on top of an existing language like Scheme.

In this dissertation, we choose the interpreter approach over the language extension approach. This choice is motivated by the fact that Flute requires drastic changes to the evaluation semantics of the existing language. Those are impossible to express using language extension facilities. Building our own interpreter gives us full control over the evaluation semantics and facilitates introducing new language semantics on demand. The interpreter approach also has other general advantages such as precise semantics of the language, and easy to experiment with new language constructs and features. Simon De Schutter observed similar trade-offs by implementing a mini variant of Flute using the language extension approach and one using the interpreter approach in his master's thesis [Sch12].

7.4 The Flute Syntax

Before discussing the internals of the Flute interpreter, let us first give a recap of its syntax. Flute's syntax extends that of Scheme [MF08]. Listing 7.1 shows the syntactic extensions to Scheme.

Listing 7.1: The concrete Flute syntax extension of Scheme.

```
1 <variable-modal>  ::= (modal (<variable>+))
2 <procedure-modal> ::= (modal (<variable>+) <expr>+)
3 <variable-mode>   ::= (mode  (<variable-modal>) <expr> <expr>+)
4 <procedure-mode>  ::= (mode  (<procedure-modal>) <expr> (<config>) <lambda>)
5 <config>          ::= (create-config <expr>+)
6 <context-event>   ::= (ctx-event)
7 <context-event>   ::= (ctx-event <expr>)
8 <uninterruptible> ::= (continuous <expr>+)
```

For brevity, here we only show the core of our extension to Scheme. A full account of the Flute syntax is provided in the Appendix. Below we summarise the purpose of each language construct:

- `modal` is for creating modals of variables and procedures.
- `mode` is for creating modes of variables and procedures.
- `create-config` is for creating interruption, resumption and state scoping strategies.
- `ctx-event` is for creating context events as reactive values, and
- `continuous` is for demarcating non-interruptible regions in a procedure body.

In the next section we describe the details of how the above constructs are supported in the interpreter.

7.5 The Flute Meta-interpreter

The Flute interpreter is implemented in a continuation-passing style (CPS) [FW08]. It explicitly passes a *continuation* parameter along with the environment. Structuring the interpreter in CPS is fundamental for realising Flute's semantics. In particular, CPS enables capturing and saving the execution context of an expression at any stage of the evaluation.

7.5.1 Architectural Overview

The general structure of the Flute meta-interpreter follows that of a classical *metacircular* evaluator with an interplay between *eval* and *apply* [AS96].[1] Figure 7.1 shows the architectural overview of the Flute meta-interpreter. It consists of the:

- **main evaluator** which is the main entry of the evaluation of a Flute program.
- **modal evaluator** that handles the evaluation of procedure modal and variable modal definitions.
- **mode evaluator** that handles the evaluation of procedure mode and variable mode definitions.
- **modal invocation evaluator** that handles the evaluation of the procedure modal application.

[1] The structure of the Flute meta-interpreter is based on the SLIP metacircular interpreter that is developed at the Software Languages Lab by Theo D'Hondt [D'H09].

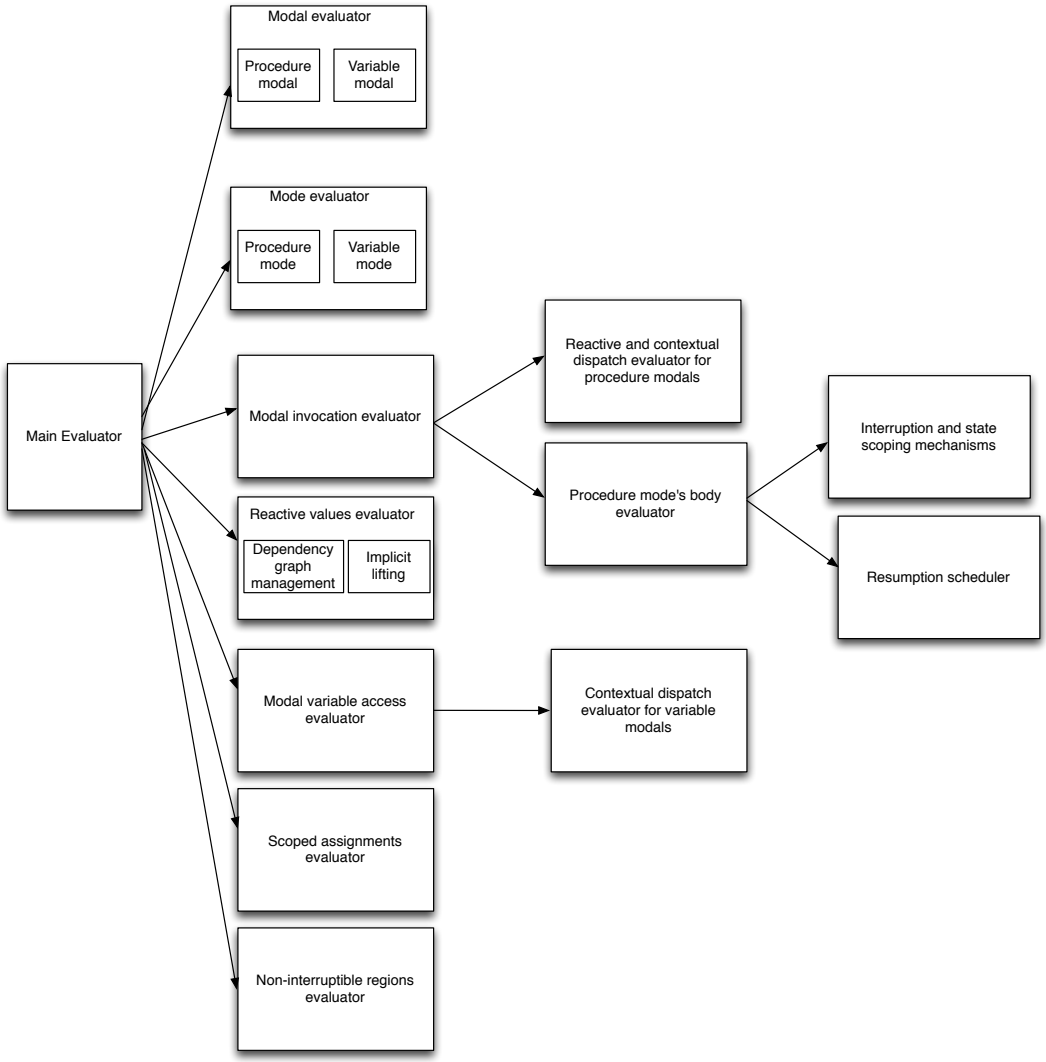

Figure 7.1: An architectural overview for the Flute meta-interpreter.

- **reactive and contextual dispatch evaluator** that handles the selection of the applicable procedure mode to execute in a reactive manner.

- **procedure mode's body evaluator** that handles the context-dependent and interruptible evaluation of a procedure mode's body expressions. In addition, it applies the appropriate interruption and state scoping strategies if the ongoing procedure evaluation is interrupted. It also handles the scheduling of suspended executions for resumption at a later moment.

- **reactive values evaluator** that handles the evaluation of reactive values. It establishes and manages the dependency graph of reactive values. In addition, it performs implicit and automatic lifting of ordinary procedures to be able to operate on reactive values.
- **variable modal access evaluator** that handles the evaluation of accessing variable modals.
- **contextual dispatch evaluator for variable modals** that handles the dispatching process for determining the value for variable modal for the current context of use.
- **scoped assignments evaluator** that handles the evaluation of state changes to variables that are shared by procedure modes.
- **non-interruptible regions evaluator** that handles the evaluation of non-interruptible regions.

In the next sections we describe the semantics of each of the above entities of the Flute meta-interpreter.

7.5.2 The Main Evaluator

At the heart of the Flute interpreter is the main evaluator that handles the evaluation of Flute programs. Its structure is as follows.

Listing 7.2: The core evaluator of the Flute interpreter

```
1  (define (eval expr continue env tailcall)
2    (cond
3      ((symbol? expr)
4       (eval-variable expr continue env))
5      ((pair? expr)
6       (let ((operator (car expr))
7             (operands (cdr expr)))
8         ((apply
9           (case operator
10            ((define)             eval-define    )
11            ((lambda)             eval-lambda    )
12            ((set!)               eval-set!      )
13            ((let)                eval-let       )
14            ((modal)              eval-modal     )
15            ((mode)               eval-mode      )
16            ((continuous)         eval-continuous)
17            ...
18            (else                 (eval-application operator)))
19          operands) continue env tailcall)))
```

```
20      (else (continue expr env)))
```

The `eval` procedure accepts the expressions to evaluate `exp`, the continuation `continue`, the environment `env`, and a Boolean value `tailcall` that indicates whether a procedure call is in a *tail position*[2]. The evaluator uses the `cond` and `case` statements to dispatch over the different ways of evaluating the expression. Each Flute construct is represented as a case clause. The interesting parts of the `eval` are the Flute constructs such as `modal`, `mode`, and `continuous`. In the remainder of this section, we discuss the evaluation of those constructs as well as the semantics of variable lookup and assignments.

7.5.3 Representing Modals

Recall from Section 6.3 that a modal is a group of related variable modes or procedure modes. In addition, it can include variables that are shared among different modes and can specify context sources that may affect the execution of those modes. The construction of a modal is handled by the `eval-modal` procedure (shown in Listing 7.3).

Listing 7.3: The evaluation of variable and procedure modal creation

```
1  ;modal creation (variable modal or proc modal)
2  ;<modal>  ::= (modal  (<variable>+))
3  ;<modal>  ::= (modal  (<variable>+) <expr>+)
4
5  (define (eval-modal . exprs)
6    (lambda (continue env tailcall)
7      (define empty-envt '())
8      (if (pair? exprs)
9          (let* ((params         (car exprs))
10                (args           (map lookup params))
11                (event-sources args)
12                (modal-envt     (bind-params params args empty-envt)))
13            (define (continue-after-modal-exprs value env-after-modal-exprs)
14              (continue (make-modal event-sources env-after-modal-exprs) env))
15            (if (pair? (cdr exprs))
16                (eval-seq (cdr exprs) continue-after-modal-exprs modal-envt tailcall)
17                (continue-after-modal-exprs '() modal-envt)))
18          (continue (make-modal '() empty-envt) env))))
```

In order to explain the evaluation of a modal expression, let us consider the following Flute program that creates a modal with a context

[2] An expression in a tail position does not require the control flow to be accumulated. This technique is known as tail call optimisation or elimination [FW08].

source `current-user` and explicit shared variables `date-range` and `display-scale`.

```
1 (modal (current-user)
2   (define date-range     2)
3   (define display-scale  4)))
```

The interpretation of the above Flute program proceeds as follows:

1. In the main evaluator, this expression matches the `modal` clause and therefore, the evaluation is handled by the `eval-modal` procedure (cf. Listing 7.2).

2. The `eval-modal` starts by extracting the context sources and creating a modal environment `modal-envt`. The modal environment contains variables that are explicitly imported (e.g., `current-user` in Listing 7.5.3). The lookup of imported variables is performed in the current environment (cf. Line 10).

3. Then the modal environment is extended with the bindings of the shared variables by evaluating the remainder of the sequence of expressions using the `eval-seq` procedure (cf. Line 16).

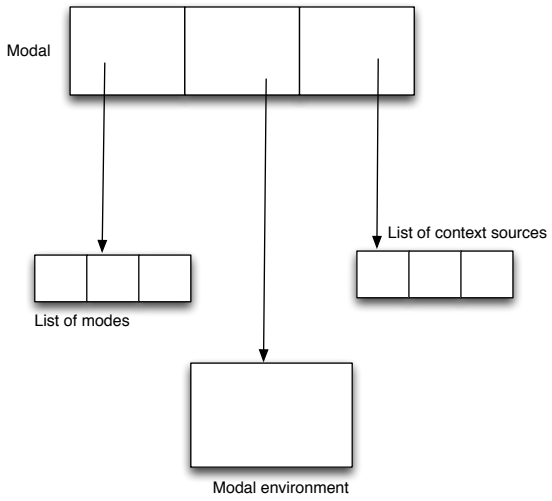

Figure 7.2: A modal representation.

4. At the end of constructing the modal environment, control is handed over to the `continue-after-modal-exprs` procedure that constructs the modal using the `make-modal` procedure (cf. Line 14). The modal is passed to the current continuation `continue` along with

the current environment `env`. A modal is represented as a vector data structure. It contains a frame of context sources and the modal environment. In addition, a modal consists of a list of modes. Figure 7.2 depicts the representation of a modal. However, the list of modes is empty at the modal creation time since there are no modes that have been added to the modal yet. In the next section, we discuss the details of creating a mode and adding it to its modal. The code excerpt below shows the definition of the `make-modal` procedure.

```
1 (define (make-modal event-sources envt)
2   (let ((modes '()))
3     (vector modal-tag modes envt event-sources)))
```

7.5.4 Representing Modes

Now that we have explained the evaluation semantics of modals, let us turn to the evaluation of modes. A mode defines a variant of behaviour (context-dependent procedure) or state (context-dependent variable) for a particular context. A context predicate is associated with each mode that constrains it to that context. Additionally, a mode definition specifies the modal to which it belongs (cf. Section 7.5.3). The evaluation of a mode creation is handled by the `eval-mode` procedure that is shown in the Listing 7.4 below.

Listing 7.4: The evaluation of a variable mode or a procedure mode.

```
1  ; mode evaluation (variable mode or proc mode)
2  ; <mode>        ::= (mode   (<modal>) <expr> <expr>+)
3  ; <mode>        ::= (mode   (<modal>) <expr> (<config>) <lambda>)
4  ; <config>      ::= (create-config <expr>+)
5
6  (define (eval-mode modal-expr pred-expr value-expr . exprs)
7    (lambda (continue env tailcall)
8      (define modal-binding (assoc (car modal-expr) env))
9      (define (continue-after-value-expr variable-value env-after-value-expr)
10       ...)
11     (define (continue-after-config mode-config env-after-value-expr)
12       ...)
13     (if (pair? exprs)
14         (eval (car value-expr) continue-after-config  env #f)
15         (eval value-expr continue-after-value-expr  env #f))))
```

Note that the `mode` construct can be used to create a variable mode or a procedure mode. Therefore, the `eval-mode` procedure handles the evaluation of variable and procedure modes. The evaluation process proceeds as follows:

1. The `eval-mode` starts by performing a lookup of the specified modal in the current environment (cf. Line 8).

2. Depending on whether the expressions `exprs` are for variable mode or a procedure mode, the evaluation process is handed over to either `continue-after-value-expr` procedure or `continue-after-config` procedure. We describe the definitions of those two procedures below.

Variable Modes. As mentioned above, the interpretation of variable modes is handled by the `continue-after-value-expr` procedure that is shown in the Listing 7.5.

Listing 7.5: The evaluation of a variable mode creation

```
1  ...belongs to the body of eval-mode in Listing 7.4 ...
2
3   (define (continue-after-value-expr variable-value env-after-value-expr)
4         (define modal (cdr modal-binding))
5         (let* ((params      '())
6                (modal-envt  (modal-env modal))
7                (mode-envt   (append modal-envt env))
8                (mode-proc   (make-proc params (list value-expr) mode-envt))
9                (mode        (make-mode pred-expr mode-proc mode-envt)))
10          (add-mode! modal mode)
11          (continue mode-proc env)))
```

The main goal of the variable mode evaluation is to create a mode and add it to the specified modal. Internally, the value for a particular variable mode is not immediately evaluated. This implies that the actual value of the mode is evaluated on demand when the variable modal it belongs to is accessed. The expression corresponding to the mode's value is represented as a procedure `mode-proc`, which is constructed using the `make-proc` procedure. The procedure's lexical environment is extended with the environment of the specified modal. The variable mode is represented as a vector data structure that consists of a predicate expression `pred-expr`, a procedure containing the mode's value expressions `mode-proc`, and the extended environment `mode-envt`. The predicate expression is used by the dispatcher to determine the applicable variable mode for the current context of use. A variable mode is added to the specified modal using the `add-mode!` procedure. Figure 7.3 depicts the representation of a mode.

Procedure Modes. The evaluation process for creating procedure modes is essentially an extension of that for creating variable modes. In addition to

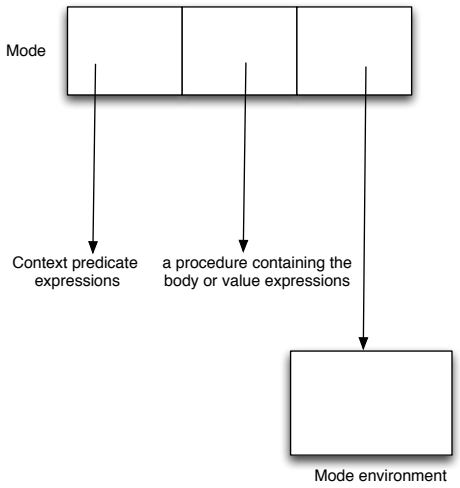

Figure 7.3: A mode representation.

a context predicate, a procedure mode also specifies the configuration options (interruption, resumption, and state scoping strategies), and a lambda expression. Listing 7.6 shows the definition of the `continue-after-config` that handles the creation of a procedure mode.

Listing 7.6: The evaluation of a procedure mode creation

```
1  ...belongs to the body of eval-mode in Listing 7.4 ...
2  (define (continue-after-config mode-config env-after-value-expr)
3     (define modal (cdr modal-binding))
4     (let* ((params          (mode-params exprs))
5            (body-exprs      (mode-exprs exprs))
6            (event-sources   (modal-event-sources modal))
7            (modal-envt      (modal-env modal))
8            (mode-envt       (append modal-envt env))
9            (mode-proc       (make-c-proc params (cons event-sources pred-expr)
10                                                  mode-config body-exprs mode-envt))
11           (mode            (make-mode pred-expr mode-proc mode-envt)))
12       (add-mode! modal mode)
13       (continue mode-proc env)))
```

The goal of this evaluation process is to construct a procedure mode and add it to the specified modal. A procedure mode is represented as a vector data structure created by the procedure `make-mode`. It consists of a predicate expression `pred-expr`, a procedure `mode-proc` and a mode evaluation environment `mode-envt`. The procedure is constructed from the parameters `params` and the body expressions `body-exprs` that are extracted from the lambda expression. In addition, it includes the context sources

`event-sources`, the configuration options `mode-config`, the mode environment. The resulting procedure mode is added to the specified modal using the `add-mode` procedure. Finally, the procedure mode is passed to the current continuation `continue` along with the evaluation environment. Unlike variable modes where the predicate is only used for selecting the applicable mode, in a procedure mode, the predicate is re-evaluated throughout the execution of the procedure body. We further discuss the semantics of the evaluation of the body of a procedure mode in Section 7.5.6. Figure 7.3 depicts the representation of a mode.

7.5.5 Evaluating Variable Modals

So far, we have discussed the *representation* of the modal and modes of variables in the interpreter. However, we have not discussed evaluation semantics of *accessing* a variable modal. Flute features uniform syntax for accessing variable modals and regular Scheme variables, for example, `(define x 10)` (cf. Section 6.4.1). However, the evaluation of variable modal references differs from that of regular Scheme variables. The difference is that accessing a variable modal yields a different value depending on the context of use. Listing 7.7 shows the definition of the `eval-modal-variable` procedure that implements the evaluation of a variable modal reference.

Listing 7.7: Evaluating variable modals.

```
1  (define (eval-modal-variable modal continue env)
2    (define modes      (modal-modes modal))
3    (define preds      (modal-preds modal))
4    (define (iterate preds modes true-count mode-proc)
5      (if (null? preds)
6          (if (= true-count end-true-count)
7              (mode-proc '() continue env #f)
8              (error "Ambiguous predicates for variable modal" true-count))
9          (let* ((head-pred      (car preds))
10                (tail-preds     (cdr preds))
11                (head-mode      (car modes))
12                (tail-modes     (cdr modes))
13                (mode-envt      (mode-env head-mode)))
14           (define (continue-after-pred value env)
15             (if value
16                 (iterate tail-preds  tail-modes (+ true-count 1)
17                          (mode-proc head-mode))
18                 (iterate tail-preds  tail-modes true-count mode-proc)))
19           (eval head-pred continue-after-pred mode-envt #f))))
20   (iterate preds modes start-true-count #f))
```

The evaluation of a variable modal proceeds as follows:

1. The evaluation starts by obtaining the list of modes and their corresponding context predicates (cf. Lines 2-3).

2. Each context predicate is then evaluated to select the right mode to execute for the current context.

3. The value for the variable modal is determined by evaluating the procedure mode whose context predicate evaluates to true `(mode-proc '() continue env #f)` (cf. Line 7).

In case there is no context predicate that evaluates to true or there are multiple context predicates that evaluate to true an ambiguous predicate exception is thrown. This semantics is appropriate for variable modal references since the expression where a variable modal reference is used expects a value to be returned. However, as we explain in the next section, the dispatching of procedure modals requires different semantics.

7.5.6 Evaluating Modal Invocations

The preceding sections have explained the details of representing procedure modals and modes. However, what we have not yet discussed is implementation for the selection of the applicable procedure mode to execute and the evaluation of the procedure modes. We now explain the evaluation of procedure modal invocations and the execution of procedure modes. When a procedure modal is invoked a reactive dispatching process is initiated to select the applicable mode to execute for the current context (cf. Section 6.5.3).

Contextual and Reactive Dispatch for Procedure Modals

The dispatching process for procedure modals is essentially similar to that of variable modals. However, unlike variables modals where the dispatching process happens only once, the dispatching process for procedure modals is repeated whenever the relevant context sources get new values. This means that even when there is no context predicate that is satisfied, there is no exception thrown. Instead the dispatcher is scheduled to be repeated on the occurrence of relevant context changes (cf. Section 6.5.3). Listing 7.8 shows the definition of the `eval-modal-dispatch` procedure that implements the dispatching of procedure modes.

Listing 7.8: Contextual dispatch for a procedure modal.

```
1 (define (eval-modal-dispatch modal continue env)
2   (define modes (modal-modes modal))
3   (define preds (modal-preds modal))
```

```
4   (define true-count-end   1)
5   (define true-count-start 0)
6
7   (define (iterate preds modes true-count mode-proc modes-after-dispatch)
8     (if (null? preds)
9         (if (= true-count true-count-end)
10            (begin
11              (modal-modes! modal modes-after-dispatch)
12              (continue mode-proc env))
13            (if (< true-count true-count-end)
14                (continue #f env)
15                (error "Ambiguous context predicates" true-count)))
16        (let* ((head-pred       (car preds))
17               (tail-preds      (cdr preds))
18               (head-mode       (car modes))
19               (tail-modes      (cdr modes))
20               (mode-envt       (mode-env head-mode))
21               (remaining-modes (cons head-mode modes-after-dispatch))
22               (head-mode-proc  (mode-proc head-mode)))
23          (define (continue-after-pred value env)
24            (if value
25                (iterate tail-preds  tail-modes (+ true-count 1)
26                         head-mode-proc modes-after-dispatch)
27                (iterate tail-preds  tail-modes true-count
28                         mode-proc remaining-modes)))
29          (eval head-pred continue-after-pred  mode-envt #f))))
30  (iterate preds modes true-count-start #f '()))
```

The entry point to the `eval-modal-dispatch` is the invocation to the `iterate` procedure that takes a list of context predicates and a corresponding list of procedure modes (cf. Lines 2-3). The evaluation of context predicates proceeds as in the case of the variable modals. The procedure mode whose context predicate evaluates to *true* is passed to the continuation `continue`, which handles the evaluation of the procedure application (cf. Line 12).

Registration of a Resumption Handler for the Dispatcher

The `eval-modal-dispatch` procedure (i.e., the dispatcher) is reified as *resumption handler* for the dispatching process that is implicitly registered as an event handler to the context sources of the modal. This makes it possible to restart the dispatching process whenever context sources receive values that may affect the context predicates. Listing 7.9 shows the implementation of the resumption handler for the dispatcher and its registration as an event handler to the relevant context sources.

Listing 7.9: Registration of a resumption handler for the dispatcher.

```
1  (define (start/restart-dispatch)
2     (eval-modal-dispatch modal continue-after-operator env))
3
4  (define (dispatcher restart-dispatch event-sources)
5    (if (ormap event? event-sources)
6        (let* ((dispatch-event (ctx-event))
7               (dispatch-thunk
8                (lambda ()
9                  (restart-dispatch))))
10         (set-thunk! dispatch-event dispatch-thunk)
11         (for-each
12          (lambda (event-source)
13            (if (event? event-source)
14                (register dispatch-event event-source)))
15          (listify event-sources)))))
```

The above code excerpt shows the reification of the `eval-modal-dispatch` as a *resumption handler* `start/restart-dispatch` (cf. Line 1). The `dispatcher` procedure implements the concerns of registering the *resumption handler* as an event-handler to the relevant context sources. By invoking the `dispatcher` procedure as `(dispatcher start/restart-dispatch event-sources)`, a new context source is created using the `ctx-event` construct. The resulting context source is bound to the `dispatch-event` name (cf. Line 6). It consists of the `dispatch-thunk` thunk that reifies the invocation to the dispatcher. It is then registered as an observer to the context sources (cf. Line 14). Whenever any of the context sources receives a new value, the thunk of the `dispatch-event` context source is invoked and hence the dispatching process is restarted. As it can be seen the above implementation for reactive dispatching depends on Flute's support for reactive values. We further discuss the evaluation semantics of reactive values in Flute in Section 7.6.

Interruptible Evaluation of Procedure Modes

As discussed in Section 6.5.4, a context predicate serves two roles: (1) it is used by the dispatcher to select the procedure mode to execute, and (2) it is used to ensure that the entire mode execution happens only in the correct context. In the interpreter, this behaviour is ensured by introducing new semantics in the evaluation of a procedure body of a mode. Listing 7.10 shows the definition of the `eval-seq` procedure that ensures that the mode's context predicates are satisfied throughout the procedure body execution. The context predicates may consist of the context predicate that is specified as

part of the mode's definition and of the context predicates that are inherited from the enclosing procedure modes (cf. Section 6.5.7).

Listing 7.10: The interruptible evaluation of a procedure body.

```
(define (eval-seq pred-expr config body-exprs args id proc-stm continue env tailcall)
  (define head                  (car body-exprs))
  (define tail                  (cdr body-exprs))
  (define event-sources         (car pred-expr))
  (define context-pred          (cdr pred-expr))
  (define resumption-mechanism  (p-true-config config))
  (define compensating-action   (p-false-config config))
  (define state-mechanism       (state-changes-config config))

  (define (continue-with-seq value env-after-seq)
    (apply-state-strategy value env-after-seq)
    (eval-seq pred-expr config tail args id proc-stm
              continue env-after-seq tailcall))

  (define (continue-after-context-pred boolean env-after-pred)
    (define (continue-after-compensating-action action-value env-after-user-action)
      (let ((state-strategy (get state-strategies state-mechanism)))
        (if state-strategy
            (state-strategy interrupted env-after-user-action)))
      (continue interrupted env-after-user-action))
    (if (and (equal? boolean #f) (interruptible? head))
        (begin
          (if (equal? resumption-mechanism resume)
              (save-execution resume-evaluation event-sources id))
          (eval compensating-action continue-after-compensating-action
                env-after-pred #f))
        (if (null? tail)
            (begin
              (apply-state-strategy completed env-after-pred)
              (eval head continue env-after-pred tailcall))
            (eval head continue-with-seq env-after-pred #f))))

  (define (seq-evaluation-entry)
    (eval context-pred continue-after-context-pred env #f))

  (seq-evaluation-entry))
```

The above procedure body evaluation differs from that of classical interpreters in that the evaluation can be interrupted at any step of the evaluation and can be resumed from where it left off at a later moment.[3] The

[3]The `eval-seq` in the Listing 7.10 is simplified to only show the main parts of the procedure's body evaluation. A complete record of the `eval-seq` implementation is provided in the Appendix.

evaluation of a procedure body is decomposed into a sequence of expressions. The context predicate expression is evaluated before the evaluation of each sequence. This ensures that the execution of the procedure happens *only* when the specified context predicate is satisfied.

The entry point for the evaluation of the procedure body is the invocation of the `seq-evaluation-entry` procedure (cf. Line 36). The context predicate is evaluated before evaluating the body expressions of a procedure (cf. Line 34). After the evaluation of the context predicate, control is handed over to the `continue-after-context-pred` procedure with the value (`true` or `false`) of the context predicate. The `eval-seq` procedure iterates over the expressions of the procedure body until either there are no more body expressions remaining or the context predicate evaluates to false. If the value of the context predicate is true, then evaluation of body expressions continues normally. However, if the value of the context predicate is false, the execution is interrupted by invoking the top level continuation `continue`. If the resumption strategy is `resume`, then the current continuation is saved by calling the `save-execution` procedure. It is not necessary to save the execution if the interruption strategy is `abort` or `restart`. Notice that the `save-execution` procedure in addition to the current execution also takes as argument the relevant context sources. This makes it possible to resume the execution when relevant context sources receive new values. The implementation of the `save-execution` procedure is shown in Listing 7.11.

Scheduling of Suspended Executions for Resumption

The scheduling a suspended execution for later resumption is handled by the `save-execution` procedure as shown below.

Listing 7.11: Scheduling suspended executions for resumption.

```
1  (define (save-execution continue-point context-sources id)
2    (if (ormap event? context-sources)
3        (let ((execution-event (ctx-event)))
4          (let ((execution-thunk
5                 (lambda ()
6                   (del saved-executions id)
7                   (del-thunk! execution-event)
8                   (continue-point))))
9            (set-thunk! execution-event execution-thunk)
10           (put saved-executions id execution-thunk))
11           (for-each
12           (lambda (context-source)
13             (if (event? context-source)
```

```
14                (register execution-event context-source) ) )
15            (listify context-sources) ) ) ) )
```

The semantics of scheduling suspended execution for resumption is reminiscent of that of reactive dispatching (cf. Section 7.5.6 Listing 7.9). The suspended execution is reified as *resumption handler* that is triggered to resume the suspended execution when relevant context sources receive new values. In the above code excerpt, `execution-event` is a resumption handler which is created using the `ctx-event` construct. It is registered as an observer to the relevant context sources (cf. Lines 11-15). When any of the context sources receives a new value, its thunk is invoked and hence the evaluation of the procedure's body expressions is resumed. Note that on resumption the context predicate is re-evaluated and if it is satisfied the execution is suspended again.

7.5.7 Evaluation of Scoped Assignments

The Flute language provides strategies to control the visibility of state changes (assignments) made during a mode execution (cf. Section 6.5.6). To this end, we augment the evaluation of assignments (`set!`) in the Flute interpreter with mechanisms to scope state changes. Listing 7.12 shows the definition of the `eval-set!` that handles the evaluation of the assignment semantics.

Listing 7.12: Evaluating assignments

```
1  (define (eval-set! variable expr)
2    (lambda (continue env tailcall)
3      (define (continue-after-expr value env-after-expr)
4        (let* ( (binding        (assoc variable env-after-expr) )
5                (active-stm     (current-stm) )
6                (write-log      (get active-stm s-write-log) )
7                (read-log       (get active-stm s-read-log) )
8                (binding-write  (assoc variable write-log) )
9                (binding-read   (assoc variable read-log) )
10               (commits        (get active-stm s-commits) ) )
11
12         (define (delay-side-effects)
13           (let ( (todo (lambda () (set-cdr! binding value) ) ) )
14             (del active-stm s-commits)
15             (put active-stm s-commits (cons todo commits) ) ) )
16
17         (define (extend-read-log)
18           (let ( (new-read-log (cons binding read-log) ) )
19             (del active-stm s-read-log)
20             (put active-stm s-read-log new-read-log) ) )
```

```
21
22          (define (extend-write-log)
23            (let* ((log-binding     (cons variable value))
24                   (new-write-log (cons log-binding write-log)))
25              (del active-stm s-write-log)
26              (put active-stm s-write-log new-write-log)))
27
28          ;if variable is already the write log, update the value.
29          (if binding-write
30              (set-cdr! binding-write value)
31              ;binding in read log but not in write log
32              ;include in the write log and update the write-binding
33              (if binding-read
34                  (extend-write-log)
35                  ;include in the read and write logs
36                  ;update the write log and add to commit queue
37                  (if binding
38                      (begin
39                        (extend-read-log)
40                        (extend-write-log))
41                      (error "inaccessible variable: " variable))))
42          (delay-side-effects)
43          (continue value env-after-expr)))
44      (eval expr continue-after-expr env #f)))
```

The technique we employ to keep track of state changes is reminiscent of that of the software transactional memory (STM) approaches [ST95, HMPJH05]. At the start of the execution of a procedure mode, a *transaction* is created for keeping track of the state changes that are performed during the execution. Thus the granularity of a transaction is on the procedure level and not based on a dedicated language construct, such as `atomic` in traditional STM approaches. The Flute constructs for controlling the visibility of state changes (i.e., `immediate, deferred, isolated`), enable the developer to tell the interpreter when state changes should be made visible (committed) to the rest of the system (cf. Section 6.5.6). However, these constructs share a common infrastructure for keeping track of state changes.

A transaction consists of a read log, a write log, and a list of commits to perform. A read log contains the values of the variables when they are first accessed. A write log contains of state changes that are performed during the execution. A list of commits consists of delayed state changes that need to be performed. The evaluation process of the assignments proceeds as follows:

1. The `eval-set!` starts by retrieving the transaction `active-stm` for the current execution, and its read log, write log, and the list of commits (cf. Lines 5-10).

2. As already mentioned above, state changes are performed to a local write log instead of the original location of the variables in the shared environment. If the variable is already in the write log, meaning it has already been previously accessed, then its value is mutated in the write log (cf. Line 29). Subsequent access to that variable during the procedure mode execution refer to its value in the write log .

3. If the variable is not in the write log but it is already in the read log (meaning that it has been accessed but has not been mutated), then the write log is extended with the new binding for the variable (cf. Line 33). Next the assignment is performed for the binding in the write log (cf. Line 40).

4. Otherwise, if the variable is neither in the read log nor in the write log, then both logs are extended with a new binding, and the assignment is performed for the binding in the write log.

5. The call to the `delay-side-effects` procedure packages the assignments to the shared environment as a nullary procedure that is then added to the list of commits to be performed at a later time (cf. Line 42).

The reason for keeping two separate logs for reads and writes is that when committing the transaction we need to perform a validation step to assert that the variables have not been changed by other modes between the time when they were first read and when the commit happens.

State Scoping Strategies

Having explained the implementation details of the transactions, we now explain the state scoping strategies (`isolated`, `deferred`, and `immediate`). As already mentioned above, all the strategies share a common transactions infrastructure. However, each strategy defines different semantics on when to make the state changes visible to other executions. Listing 7.13 shows the definition of the strategies.

Listing 7.13: State scoping strategies.

```
1 ;commit and continue execution --
2 ;validate changes and commit or abort
3 (define (commit-changes status env-after-seq)
4   (let* ((active-stm (top active-stms))
5          (commits    (get active-stm s-commits))
6          (read-log   (get active-stm s-read-log)))
7     ...
```

```
8       (if (validate read-log env-after-seq)
9           (commit (reverse commits))
10          (abort-execution))))
11
12  ;state:isolated strategy
13  (define (isolated-strategy status env-after-seq)
14    (ignore-changes status))
15
16  ;state:deferred strategy
17  (define (deferred-strategy status env-after-seq)
18    (if (equal? status completed)
19        (commit-changes status env-after-seq)
20        (ignore-changes status)))
21
22  ;state:immediate strategy
23  (define (immediate-strategy status env-after-seq)
24    ;this is opposite of isolation
25    ;i.e., on interruption or completion
26    (commit-changes status env-after-seq))
```

The `commit-changes` procedures implements the logic of committing a transaction. It implements a two-step process that consists of a validation step and a commit step. The validation step asserts that variables in the read log have the same values as in the current environment. If they have the same values it implies that no other execution has modified those variables. Therefore, the commit step of the delayed assignments is performed. Otherwise, the commit process is aborted. The decision on when to commit state changes is different for each of the state scoping strategies.

Isolated strategy ensures that state changes remain visible only local to the execution and are not committed to the shared environment. The `isolated-strategy` procedure defines the logic of the `isolated` state scoping strategy. The transaction is discarded once the procedure execution completes (cf. Lines 13-14).

Deferred strategy ensures that the state changes are committed when the procedure execution completes. The `deferred-strategy` procedure in Listing 7.13 implements the logic of the `deferred` state scoping strategy. The flag `status` indicates whether a procedure execution has been interrupted or has completed. If the procedure execution has been interrupted, then the state changes are not committed. However, the transaction log is retained for later use when the execution is resumed. If the procedure execution completes, then the state changes are committed by calling the `commit-changes` procedure (cf. Lines 17-20).

Immediate strategy, unlike the deferred strategy, commits the state changes on interruption or completion. In Listing 7.13, the `immediate-strategy` procedure implements the logic of the `immediate` state scoping strategy (cf. Lines 23-26).

7.5.8 Representing Uninterruptible Regions

As discussed in Section 6.5.8, Flute enables the developer to demarcate certain regions in a procedure body as uninterrupted using the construct `(continuous <exprs>)`. Listing 7.14 shows the definition of the `eval-continuous` procedure that implements the evaluation of uninterruptible regions.

Listing 7.14: Evaluating uninterruptible regions

```
1 (define (eval-continuous . exprs)
2   (lambda (continue env tailcall)
3     (let* ((params '())
4            (args   '())
5            (proc (make-proc params exprs env)))
6       (eval (cons proc args) continue env #f))))
```

The evaluation converts the expressions `exprs` into a nullary procedure that is then immediately scheduled for evaluation. Since this procedure has no associated context predicate, the evaluation of its body (i.e., the expressions `exprs`) will complete without interruption.

7.6 Semantics of Reactive Values in Flute

As discussed in Section 6.7.2, context sources are represented as reactive values. In this section we describe exact semantics of of reactive values in Flute. Recall that a reactive value is created using the `ctx-event` construct. For instance, a context source for representing the current GPS coordinates is defined as a reactive value as follows.

Listing 7.15: Defining context sources as reactive values.

```
13 ;Creating  a context source
14 (define gps-coordinates (ctx-event))
```

Reactive values do not require sophisticated evaluation machinery like that of procedure or variable modes. It is possible to implement the constructs for reactive values as native Flute procedures instead of special forms. At

startup, the environment for the Flute interpreter is initialised with a list of native Flute procedures that can be used in Flute programs.

7.6.1 Evaluation of Defining Reactive Values

Listing 7.16 shows the definition of the `cps-ctx-event` procedure that implements the creation of a reactive value.

Listing 7.16: Evaluating the creation of reactive values.

```
1  ;native for creating a reactive value
2  (define (cps-ctx-event expr continue env tailcall)
3    (define meta-ctx-event ctx-event)
4    (if (pair? expr)
5        (continue (meta-ctx-event (car expr)) env)
6        (continue (meta-ctx-event) env)))
```

The `cps-ctx-event` procedure takes as argument the expression `expr` that represents an optional initial value for a reactive value. The creation of a reactive value is delegated to a non-cps iScheme procedure `ctx-event`. In the Flute environment the `cps-ctx-event` is mapped onto the `ctx-event` procedure by adding a binding to the global environment (i.e., `(cons 'ctx-event cps-ctx-event)`. The definition of the iScheme `ctx-event` procedure is shown in Listing 7.18.

7.6.2 Evaluation of Updating Reactive Values

Recall that Flute provides the `update-value!` construct for updating the value of a reactive value (cf. Section 6.7.1). The `update-value!` construct is particularly used to acquire (non-reactive) low-level data from external sensors (cf. Section 6.7.2). In the Flute interpreter, the `update-value!` construct is implemented as follows.

Listing 7.17: Updating a reactive value.

```
15 (define (cps-update-value! expr continue env tailcall)
16   (define event (car expr))
17   (define new-event (cadr expr))
18   (define meta-update-value! update-value!)
19   (meta-update-value! event new-event)
20   (continue event env))
```

The `update-value!` construct is mapped onto the `cps-update-value!` procedure above. The `cps-update-value!` procedure takes as argument the expression `expr` that consists of a

reactive value and the value to update the reactive value to. The updating of a reactive value is delegated to the non-CPS iScheme procedure `update-value!`.

7.6.3 Representing Reactive Values

At the meta level, a reactive value is represented as vector data structure (as depicted in Figure 7.4). It consists of the following:

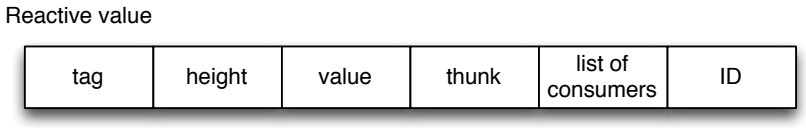

Figure 7.4: A representation of a reactive value.

- the tag that is used by the Flute interpreter to check whether a value is a reactive value or not.
- the height that is used for ordering the dependency graph of reactive values in order to ensure that propagation of changes happen consistently without glitches. Initially, the height of a freshly constructed reactive value that does not depend on any other reactive values is 0.
- the value of the reactive value.
- the thunk that is invoked to update the reactive value's value whenever the reactive values it depends on receive new values.
- the list of consumers that depend on the reactive value's value. Initially, the list of the consumers is empty. The list of consumers are scheduled for update whenever the reactive value receives a new value.
- the identifier for the reactive value.

Listing 7.18 shows the definition of the `ctx-event` iScheme procedure that is responsible for creating reactive value data structure.

Listing 7.18: Creating reactive values.

```
1 (define (ctx-event . value)
2   (define (new-event)
3     (define thunk (lambda () 'no-value))
4     (vector event-tag initial-height undefined thunk '() (getid)))
5
```

```
6   (let ((self (new-event)))
7     (if (and (pair? value) (event? (car value)))
8         (begin
9           (register self (car value))
10          (set-thunk! self
11                      (lambda ()
12                        (update-value! self (current-value (car value)))))
13          (update-value! self (current-value (car value))))
14        (if (pair? value)
15            (update-value! self (car value))))
16    self))
```

The `ctx-event` procedure takes as argument an optional initial value for a reactive value. If there is no initial value provided, a new reactive value is constructed using the `new-event` procedure (cf. Line 6) and its initial value is `undefined`. If an initial is provided, the value of the reactive value is set to that value. The initial value can either be non-reactive value (e.g., `1`) or a reactive value. In case the initial value is a reactive value, it is necessary to establish a dependency between the two reactive values. Such a dependency is established by calling the `register` procedure (cf. Line 9). By establishing a dependency between the two reactive values, it is ensured that the value of the new reactive value is updated whenever the value of the reactive value it depends on changes. The thunk of the new reactive value reifies the action of updating the value of the reactive value. The `current-value` procedure is used to obtain the current value of a reactive value. The implementation of the `current-value` procedure is as follows:

Listing 7.19: Obtaining the current value of a reactive value.

```
1   (define (current-value any)
2     (if (event? any)
3         (vector-ref any value-idx)
4         any))
```

The `current-value` procedure takes as argument a reactive value data structure and retrieves the its current value at the index `value-idx`.

7.6.4 Establishing a Dependency between Reactive Values

One of the key semantics of reactive values is that the language runtime automatically tracks dependencies among reactive values. For instance, as discussed in Section 7.6.3, creating a new reactive value in terms of another requires registering a dependency between the two reactive values. We refer

to the reactive value that depends on another reactive value as a *consumer*, and the reactive value that is depended on by another reactive value as the *producer*. Listing 7.20 shows the `register` procedure that implements the logic of establishing a dependency between reactive values.

Listing 7.20: Establishing a dependency between reactive values.

```
(define (register consumer producer)
  (if (not (consumer-exists? consumer producer))
      (let* ((consumer-height (event-height consumer))
             (producer-height (event-height producer))
             (max-height (max consumer-height producer-height)))
        (update-height! consumer (+ 1 max-height))
        (new-consumer! producer consumer))))
```

The `register` procedure takes as argument a consumer reactive value and a producer reactive value and establishes a unidirectional dependency between the producer and the consumer. Each reactive value maintains a list of its consumers. As discussed in Section 7.6.3 a reactive value has a height. When establishing a dependency between reactive values, the consumer's height is updated to be higher than that of the producer. This means that the consumer's height is always higher than that of any of its producers. When scheduling reactive values for updating, the height is used to topologically sort a graph of reactive values. This ensures that the processing of updates happens in the correct order (without glitches)(cf. Section 6.7.1). This glitch avoidance technique is similar to that of other functional reactive programming approaches such as FrTime [CK06] and Flapjax [MGB+09]. Note that the dependency graph must be acyclic in order to avoid non-terminating propagation of changes.

7.6.5 Supporting Implicit Lifting

Recall from Section 6.7.1 that it is possible to apply native iScheme procedures such as `<`, `+` and user defined procedures to reactive values. For instance, one can write a reactive program in Flute as follows.

Listing 7.21: A reactive program that requires implicit lifting.

```
(define x (ctx-event 1))
(define y (< x 1))
```

The iScheme < native procedure expects numbers as arguments and applying it to a reactive value x would generate a type error. When the Flute evaluator encounters such an expression, the procedure is *implicitly lifted* to be able operate on the reactive value and returns a new reactive value whose value

at any moment is the result of applying that procedure to its arguments. Listing 7.22 shows the implementation of the procedures that handle the lifting of ordinary iScheme procedures.

Listing 7.22: Lifting of procedures to operate on reactive values.

```
1  (define (event-values events)
2    (map current-value events))
3
4  (define (frpify proc)
5    (lambda args
6      (let ((arguments (event-values args)))
7        (apply proc arguments))))
8
9  (define (lift proc)
10   (lambda args
11     (let* ((new-event (ctx-event))
12            (frp-proc  (frpify proc))
13            (thunk (lambda ()
14                     (update-value! new-event
15                                    (frp-proc args)))))
16       (set-thunk! new-event thunk)
17       (thunk)
18       (for-each
19        (lambda (event)
20          (if (event? event)
21              (register new-event event)))
22        args)
23       new-event)))
```

If any of the arguments of a procedure is a reactive value, the evaluator automatically lifts the procedure using the `lift` procedure above. First, a new reactive value is created (cf. Line 11). The ordinary procedure is then *lifted* to a procedure that can operate on reactive values using the `frpify` procedure (cf. Line 12). The `frpify` procedure obtains the values of the reactive values using the `event-values` procedure and applies the ordinary iScheme procedure. The thunk reifies the action of updating the value of the newly constructed reactive value to the result of applying the lifted procedure to the arguments. Finally, the reactive value is registered as a consumer of any reactive values that are in the arguments of the procedure (cf. Lines 18-22). Whenever any of the reactive values receives a new value the thunk is called again and the value of the reactive value is updated.

7.7 Chapter Summary

In this chapter, we have described exact semantics of the Flute language through a meta-interpreter. We chose to implement the Flute language as an interpreter because it gives us full control over the evaluation of every language detail. We have presented the interpreter starting from the general evaluator and then the individual evaluators for each core construct of Flute. The interpreter is implemented in a continuation passing style, which facilitates the task of capturing and saving the procedure execution at any evaluation step.

Chapter 8

Building Reactive Context-aware Applications Using Flute

Contents

8.1 Introduction

This chapter validates the ICoDE model instantiation in the Flute programming language. Recall from Section 2.2 that we introduced a visionary scenario – dubbed *BainomuAppies* in Kampala – that was used to *motivate reactive* context-aware applications. In the *BainomuAppies* scenario, buses and minibuses in Kampala are equipped with an onboard digital platform that runs a suite of applications. The current application running as well as the nature of the information shown depends on contextual information such as the geolocation of the bus, the proximity of other buses and certain stops, and the identity of the passengers on the bus at a certain moment in time. Given the large scale and the required infrastructure of the scenario it is not

feasible to realise it within the scope of a Ph.D. dissertation. We therefore consider a variant of the *BainomuAppies* in Kampala scenario that was more feasible to implement, yet equally representative for *reactive* context-aware applications.

We have implemented a prototype mobile platform called *iFlute Platform* on which context-aware applications can be deployed. This scenario exhibits characteristics similar to those of the *BainomuAppies in Kampala* scenario (cf. Section 2.3). We use Flute as the programming language to develop context-aware applications that are deployed on the iFlute platform. These include a context-aware calendar application, a context-aware printer assistant, and a context-aware task assistant. As in the *BainomuAppies in Kampala* scenario, the currently running application and its behaviour depend on the current context of use. For each application we present its implementation and evaluate Flute's language constructs against the requirements that should be satisfied by a programming language for *reactive* context-aware applications (cf. Section 2.6). We conclude this chapter by comparing Flute with existing approaches. In particular, we compare the implementation of some of the examples in the Flute programming language with that of first-class continuations – one of the few approaches that provide support for (explicit) interruptions and resumptions (cf. Section 3).

8.2 The iFlute Platform

The iFlute platform consists of a suite of applications for different tasks (each task corresponds to a behaviour for a different context) in the context of a personal assistant. To alleviate the burden of manually selecting which application and behaviour to run for the current task, the iFlute platform is enhanced with context-awareness to automatically present to the user the appropriate behaviour for the task at hand. When there is a context change, the running application's behaviour is promptly interrupted and replaced by a new behaviour that is appropriate for that context. The previously interrupted application's behaviour can be resumed from where it left off at a later moment when the user goes back to the previous context. Figure 8.1 shows the screenshot of the iFlute platform running on an iOS device.

Example applications that have deployed on the iFlute platform include a context-aware calendar application, a context-aware printer assistant, and a context-aware task assistant. Like in the BainomuAppies scenario, the current application and its behaviour depends on the context of use. Below we briefly describe each of the applications:

Figure 8.1: The iFlute platform running on a tablet device.

***Kalenda:* A Context-aware Agenda Application.** The first example is the context-aware calendar application that we used as a running example for explaining the Flute programming language (cf. Section 6.2). *Kalenda* automatically launches to show the agenda items whenever the user moves within range of his/her workplace. The agenda items may include the user's private appointments (e.g., family events or a doctor appointment) that should be displayed only to the device owner and public items (e.g., workplace meetings or bank holidays) that can be seen by anyone. Therefore, when the owner is not the one using the device, it can dynamically adapt to show only the agenda items that are public. For instance, suppose that a user who is browsing through his/her private calendar items temporarily hands over the device to a coworker. The calendar application should adapt immediately to show only public items and adapt the display properties (e.g., font size or colour) to match the coworker's preferences. Furthermore, if the coworker gives the device back to the owner, the calendar application should immediately restore the owner's previous view of the agenda items.

***Pulinta:* A Context-aware Printer Assistant.** The *Pulinta* application automatically launches when the user is in a printer room or is nearby a printer. It provides functionalities for monitoring a printer's status (e.g., toner and paper levels) and managing printing tasks. The printing tasks are enhanced with context-awareness to dynamically adapt the printing of sensitive documents on a shared printer. For instance, when the user is printing sensitive documents from his/her mobile device and another person walks into the printer room, the printing task is interrupted promptly. The interrupted printing task is resumed from where it left off when that person leaves.

***Tasiki:* A Context-aware Task Assistant.** The third example is a task assistant application that assists the user to perform routine daily tasks such as managing phone calls and processing emails. For a person with a busy schedule it can hectic to keep track of the people that need to be called during the course of the day. Moreover, for certain people it may be appropriate to use a specific communication technology (e.g., Skype, teleconferencing equipment or a state-of-the-art phone system in a car). At the beginning of the day, the *Tasiki* application enables the user to specify the people to call, the required technology, when and where it is appropriate to initiate phone calls to certain contacts. The application uses this information to automatically initiate phone calls to certain contacts and connect to the right technology whenever the user is in the right context and there are

contacts that need to be called. For instance, it may be convenient to make phone calls to co-workers when the user is at his/her workplace's conference room while for business people a modern phone system in car might be convenient. Thus, when the user is in a conference room at his/her workplace, the application checks if there are any co-workers contacts to call and offers to initiate the phone calls. Similarly, when the user is in his/her car the application is automatically launched and offers to initiate phone calls to the business contacts, if any. The application always remembers the next contacts to call and automatically resumes from where it left off as the user moves about.

8.3 Implementing the iFlute Platform

We have implemented a prototype of a mobile platform called the *iFlute Platform* where *reactive* context-aware applications can be deployed. The screenshot of the platform running on an iOS device is shown in Figure 8.1. The GUI concerns of the platform are implemented in iScheme. In this chapter, we will focus on the implementation of the context-dependency concerns. A complete record of the iFlute platform implementation is provided in the appendix.

The context-dependency concerns of the iFlute platform are structured around modals, modes, and context sources as follows.

Modal	**Mode**	**Context predicate**	**Context source**
flute-apps	calendar-assistant	office?	current location
	printing-assistant	printer-room?	
	task-assistant	at-conference-room-or-car?	

Table 8.1: Modals and modes of the iFlute platform.

- Context-aware applications on the iFlute platform belong to the modal `flute-apps`. The context source for the modal is the current location, which is derived from the GPS sensor. Context sources are defined as reactive values.

- Each context-aware application is represented as a mode that belongs to the `flute-apps` modal.

- Each context-aware application is associated with a context predicate that constrains it to a specific context: the *Kalenda* application is

constrained to run only when the user is in his/her office, the *Pulinta* application is constrained to run only when the user is in a printer room, and the *Tasiki* application is constrained to run only when the user is at his/her workplace's conference room or car. Figure 8.2 depicts an overview of the iFlute platform and the context-aware applications that are deployed on it.

Implementation of Context Sources as Reactive Values. The implementation of the location context source for the iFlute platform is shown below.

Listing 8.1: Defining context sources as reactive values in Flute

```
1 (define gps-coordinates (ctx-event))
2 (define location        (gps->location gps-coordinates))
3
4 (CURRENT-LOCATION
5  (lambda (latitude longitude)
6    (update-value! gps-coordinates
7                   (cons latitude longitude))))
```

The context source `gps-coordinates` as a reactive value using Flute's `ctx-event` construct. Its values are raw latitude and longitude location coordinates. However, the iFlute platform requires high-level context information (e.g., the location as office or printer room) to decide on which application to run. The procedure `gps->location` transforms GPS coordinates into the desired high-level location names. Since `gps->location` operates on a reactive value `gps-coordinates`, its return value is also a reactive value that is bound to the variable `location`. As such, the `gps->location` procedure is automatically re-evaluated whenever `gps-coordinates` receives a new value. Hence, the value of the `location` variable is also updated. The acquisition of (non-reactive) raw location coordinates is accomplished using the `CURRENT-LOCATION` construct, which is provided by iScheme (cf. Section 5.5).

Implementation of the Context Predicates. Each application on the iFlute platform is associated with a context predicate that determines its applicability to the current context of use. Listing 8.2 shows the definition of the context predicates. The `office?` predicate evaluates to true when the current location is office, the `printer-room?` predicate evaluates to true when the current location is printer room, while the `at-conference-room-or-car?` predicate evaluates to true when the

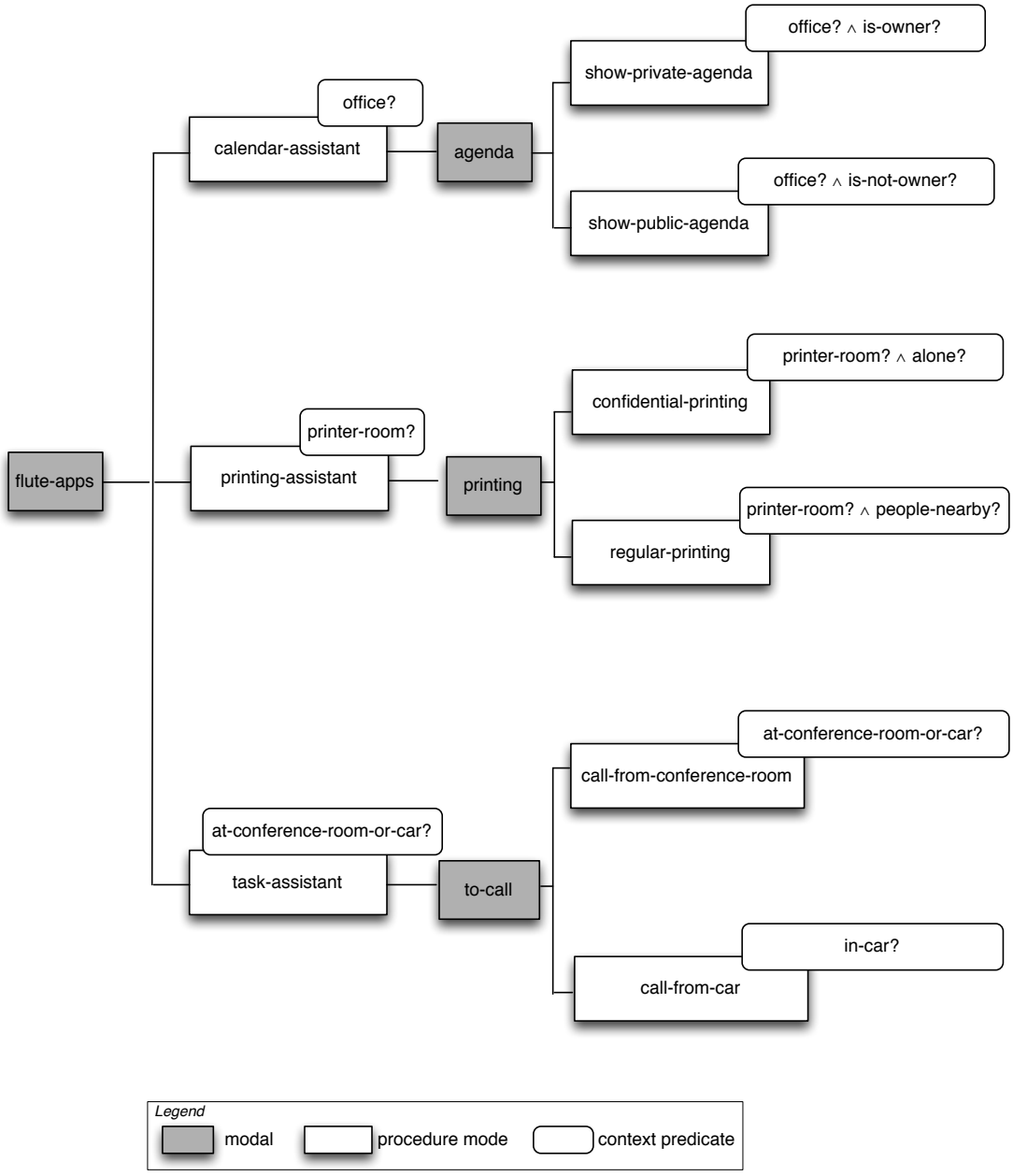

Figure 8.2: An overview of the iFlute platform and the context-aware applications that are deployed on it.

current location is conference room or car.[1] We assume that the context predicates are mutually exclusive (i.e., at most, only one context predicate can be satisfied).

Listing 8.2: Implementation of the context predicates.

```
(define (office? user-location)
  (equal? user-location office))

(define (printer-room? user-location)
  (equal? user-location printer-room))
```

Implementation of the iFlute Platform using Modals and Modes. The iFlute platform is structured as a modal grouping several modes, each corresponding to an application. The code listing below depicts how the iFlute platform is implemented using the abstractions of the Flute programming language.

Listing 8.3: Implementation of the iFlute platform in Flute

```
;The iFlute platform modal for the apps belong to the platform
(define flute-apps (modal (location)))

(define config
  (create-config suspend resume isolated))

;Kalenda application defined as a mode belonging to the flute-apps modal
(define calendar-assistant
  (mode (flute-apps)
    (office? location)
    (config)
    (lambda ()
      ;...cf. Section 8.4 for complete implementation
      )))

;Pulinta application defined as a mode belonging to the flute-apps modal
(define printing-assistant
  (mode (flute-apps)
    (printer-room? location)
    (config)
    (lambda ()
    ;... cf. Section 8.5 for complete implementation
    )))

;Tasiki application defined as a mode belonging to the flute-apps modal
```

[1] In Listing 8.2, `office` and `printer-room` are globally defined variables whose values are symbols `'office`, and `'printer-room` respectively.

```
26 (define task-assistant
27   (mode (flute-apps)
28     (at-conference-room-or-car? location)
29     (config)
30     (lambda ()
31     ;...cf. Section 8.6 for complete implementation
32     )))
33
34 ;starting all apps on the iFlute platform
35 ...
36 (flute-apps)
```

The modal `flute-apps` represents the iFlute platform to which applications (modes) can be added. The `flute-apps` modal specifies `location` as the context source (cf. Listing 8.1). The `flute-apps` modal defines a grouping entity for the platform's applications: a context-aware calendar application, a printing assistant application, and a context-aware task assistant application. Each application is defined as a mode. The `calendar-assistant` mode represents the context-aware calendar application, the `printing-assistant` mode represents the context-aware printing assistant application, while the `task-assistant` mode represents a context-aware task assistant application. Each application mode is associated with context predicates (cf. Listing 8.2) to specify the context in which the mode is allowed to execute. Additionally, each mode is specified with the configuration option `config` that specifies the interruption strategy `suspend`, the resumption strategy `resume`, and the state scoping strategy `isolated`. The `suspend` and `resume` strategies ensure that each application is suspended when its associated context predicate is no longer satisfied and is resumed when the context predicate is later satisfied again. The `isolated` strategy ensures that the state changes of each application remain isolated from the rest of the system.

When the iFlute platform starts up, the applications are deployed (cf. Figure 8.1) and the `flute-apps` modal is invoked. By invoking the `flute-apps` modal, the dispatching process is initiated to select the application that is appropriate for the current context of use. The application to run depends on the context predicate that evaluates to true. In case there is no context predicate that evaluates to true, no application is launched. However, when the context source `location` receives a new value, the dispatching process is repeated to select the matching application. Thus as the user moves about, the running application may become suspended or resumed to ensure that the running application matches the current context. Note that each application consists of context-dependent behavioural variations that we discuss in detail in the next sections.

Evaluation

From the above implementation of the iFlute platform, we make the following observations that showcase the strengths of the Flute language. We align each strength with the requirements that should be satisfied by a programming language for *reactive* context-aware applications (cf. Section 2.6).

- The modal and mode abstractions of the Flute language enables defining the iFlute platform as an extensible group of applications. The `flute-apps` modal makes it possible to add new modes of applications whenever required. Each application is associated with a context predicate that is used to select the appropriate application to run for the current context of use. As the selection of the application to run is managed by Flute's runtime, there is no need for explicit context checks (cf. **Requirement** *R*.4 *Contextual dispatch*). Furthermore, the specified context predicate is re-evaluated continually throughout the execution of a mode's procedure body. Hence, the application's execution is always constrained to the correct context (cf. **Requirement** *R*.2 *Context-dependent interruptions*).

- By defining context sources as reactive values, it is possible to use them in context predicates without requiring the use of explicit event handlers. For instance, the `location` context source which denotes the current user location is used in the predicates for the modes. Also, Flute's language runtime ensures the value of `location` is automatically updated whenever the value of `gps-coordinates` changes. Without such support the developer would have to manually ensure that `location` is updated whenever `gps-coordinates` receives a new value (cf. **Requirement** *R*.1 *Chained context reactions*). Furthermore, the dispatcher is notified whenever the relevant context sources receive new values. This automatically re-triggers the dispatching process again to select the appropriate application to run for the current context (cf. **Requirement** *R*.5 *Reactive dispatch*).

- The configuration option `config` enables the developer to specify the interruption, resumption, and state scoping strategies (cf. **Requirements** *R*.2 *Context-dependent interruptions*, *R*.3 *Context-dependent resumptions*, and *R*.6 *Reactive scope management*). For instance, the context-aware calendar application specified with the `suspend` interruption strategy and the `resume` strategy. Flute's language runtime uses these developer-specified strategies in ensuring that the execution state is preserved between interruptions. The developer does not need

to worry about when to interrupt or resume an execution. Moreover, each application's mode is specified with the `isolated` state scoping strategy which ensures that state changes remain local to the application. Without such language support the developer would have to control manually the visibility of state changes, which is not trivial and may lead to an inconsistent execution environment.

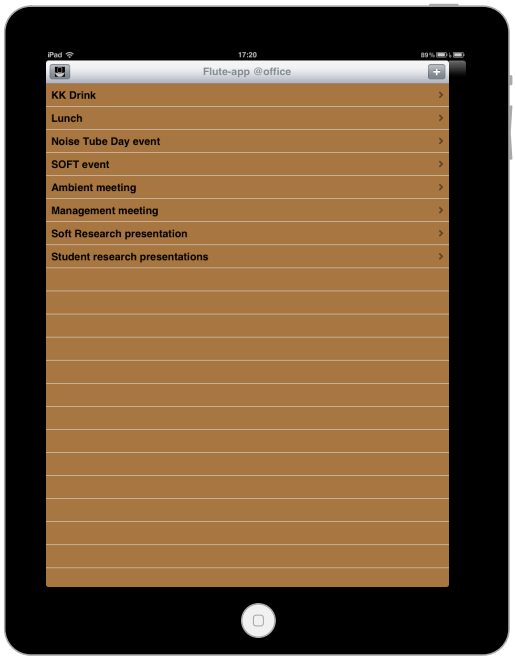

Figure 8.3: The *Kalenda* application executing in the public mode.

8.4 Implementing the *Kalenda* Application

One of the applications deployed on the iFlute platform is the *Kalenda* application. As introduced in Section 8.2, the *Kalenda* application adapts its behaviour to show private or public agenda items depending on whether the device is being used by the owner or not. Additionally, the display properties such as background colour and font size vary depending on the user. To implement the *Kalenda* application in Flute, we structure it as follows.

- The procedure modal of the *Kalenda* application is the `agenda` modal. This modal groups together behavioural modes of the *Kalenda* application. The modal specifies the current user of the device as the context

Modal	Mode	Context predicate	Context source
agenda	show-private-agenda	is-owner?	current-user
	show-public-agenda	is-not-owner?	
bg-colour	grey colour	is-owner?	
	brown colour	is-not-owner?	
calendars	personal calendars	is-owner?	
	public calendars	is-not-owner?	

Table 8.2: Modals and modes of the *Kalenda* application.

source that is used in the context predicates of the modes. In addition, it specifies variables that are shared by all its modes. These include variables for display scale and date range that are used to fine-tune the display properties of the calendar.

- There are two procedure modes for the context-dependent behavioural variations of the *Kalenda* application: The `show-private-agenda` mode that is executed when the device is being used by the owner, and the `show-public-agenda` mode that is executed when the user is not the device owner.

- There are two variable modals: `bg-colour` variable modal that stores a different value (colour) depending on the context (i.e., whether the current user is owner or not), and the `calendars` variable modal that stores a different list of calendars depending on the context.

Listing 8.4 shows the implementation of the *Kalenda* application in Flute. The GUI concerns of the application and the interaction with the native calendar APIs of the iOS are implemented using the symbiosis provisions of iScheme (cf. Section 5.4).

Listing 8.4: The Implementation of the *Kalenda* in Flute

```
1  ;definition of the contetx source as a reactive value
2  (define current-user (ctx-event))
3
4  ;context predicates
5  (define (is-owner? current-user)
6    (equal? current-user device-owner))
7
8  (define (is-not-owner? current-user)
9    (equal? current-user not-device-owner))
10
11 ;calendar application defined as a mode belonging to the flute-apps modal
```

```
12 (define calendar-assistant
13   (mode (flute-apps)
14     (office? location)
15     (config)
16     (lambda ()
17       (define calcontrollername "Flute Calendar Manager")
18       (define UIColor           (OBJC-CLASS UIColor))
19       (define calcontroller     (get iFluteappcontrollers calcontrollername))
20       (define tableView         (OBJC-SEND calcontroller tableView))
21
22       (define (matching-events filter-pred)
23         (OBJC-SEND eventstore eventsMatchingPredicate: filter-pred))
24
25       (define (refresh-tableview colour)
26         (OBJC-SEND calcontroller setBackgroundColor: colour)
27         (OBJC-SEND tableView reloadData))
28
29       ;variable modal for calendar objects
30       (define calendars (modal (current-user)))
31
32       ;variable mode - personal calendars
33       (mode (calendars)
34         (is-owner? current-user)
35         (get-personal-calendars))
36
37       ;variable mode - public calendars
38       (mode (calendars)
39         (is-not-owner? current-user)
40           (get-public-calendars))
41
42       ;variable modal for background colour
43       (define bg-colour (modal (current-user)))
44
45       ;variable mode - public agenda colour
46       (mode (bg-colour)
47         (is-not-owner? current-user)
48          (OBJC-SEND UIColor brownColor))
49
50        ;variable mode - private agenda colour
51        (mode (bg-colour)
52          (is-owner? current-user)
53           (OBJC-SEND UIColor grayColor))
54
55      ;procedure modal for agenda behavioural variations
56      (define agenda (modal (current-user)
57         (define date-range            2)
58         (define display-scale         4)))
59
60       ;procedure mode - showing private agenda behaviour
```

```
61        (define show-private-agenda
62          (mode (agenda)
63            (is-owner? current-user)
64            (config)
65            (lambda ()
66              (define cals-array (list->NSMArray calendars))
67              (define filter-pred (create-pred cals-array date-range))
68              (define day-events      (matching-events filter-pred))
69              (define eventslist      (OBJC-INSTANCE NSMutableArray))
70              (set-events-list! eventslist)
71              (set-default-cal! (car calendars))
72              (add-events-to-eventslist eventslist day-events)
73              (set! display-scale 8)
74              (scale display-scale)
75              (refresh-tableview bg-colour))))
76
77        ;procedure mode - showing public agenda behaviour
78        (define show-public-agenda
79          (mode (agenda)
80            (is-not-owner? current-user)
81            (default-config)
82            (lambda ()
83              ;... event filtering and agenda items setup
84              (set-events-list! eventslist)
85              (set-default-cal! (cdr calendars))
86              (refresh-tableview bg-colour))))
87
88
89        (agenda)))
```

In the above implementation, `calendar-assistant` is a mode definition for the *Kalenda* application that belongs to the `flute-apps` modal (cf. the implementation of the iFlute platform in Section 8.3). It is associated with the context predicate `office?` to specify that it is launched when the user is in his/her office. The `office?` context predicate operates on the context source `location` that was specified as part of the definition of the `flute-apps` modal (cf. Listing 8.3). The `current-user` context source is a reactive value which is defined using the `ctx-event` construct. At any given time, the value of the `current-user` denotes the current user of the device (i.e., the owner or not the owner). In this scenario, we use the device orientation sensor of the iOS to simulate the different device users. But in the future, we can imagine the use of NFC-enabled wristwatches to detect the identity of a user. The `is-owner?` and `is-not-owner?` context predicates evaluate to true or false depending on whether the current device user is the owner or not. As with the iFlute platform implementation, we assume that the context predicates are mutually exclusive (i.e., at most one

context predicate can be satisfied).

The two variable modals `calendars` and `bg-colour` respectively store a different calendar list and background colour depending on the current context of use. The `calendar` variable modal consists of two modes: the first mode is associated with the `is-owner?` context predicate and evaluates to the list of personal calendars, while the second mode is associated with the `is-not-owner?` context predicate and evaluates to the list of public calendars. In this scenario, the calendars are retrieved from the iOS native calendar store using the iScheme constructs. The `bg-colour` variable modal also consists of two modes: one that evaluates to the colour grey, and the other evaluates to the colour brown. The colour values are retrieved from the iOS APIs using the iScheme symbiosis constructs. The `agenda` modal groups together different modes of the calendar. It specifies the `current-user` as the context source. In addition, it includes the shared variables `date-range` and `display-scale` that specify the range of agenda items to show and the display scale for the calendar.

The `show-private-agenda` mode implements the context-dependent behaviour for showing private agenda items. It is associated with the context predicate `is-owner?`, which ensures that the private agenda items are shown only when the device is being used by the owner. The `show-private-agenda` mode specifies its modal as `agenda`. Note that since the `show-private-agenda` mode is defined within the `calendar-assistant` mode, it inherits the context predicate `office?` that is associated with the `calendar-assistant` mode (cf. propagation of context predicates in Section 6.5.7). This implies that both `office?` and `is-owner?` context predicates should be satisfied during the execution of the `show-private-agenda` mode. In addition, it specifies the configuration option as `config`. As defined in Listing 8.3, the `config` configurations specify `suspend`, `resume`, and `isolated` as the interruption, resumption and state scoping strategies. The `suspend` and `resume` strategies ensure that the device owner always resumes from the same point as before, in case the device owner gives the device to another user. The `isolated` strategy ensures that state changes remain local to the `show-private-agenda` mode. Thus the state change that is performed on the `display-scale` variable (i.e., `(set! display-scale 8)`) remains only visible to the `show-private-agenda` mode.

As `calendars` and `bg-colour` are variable modals, they evaluate to values of modes that are guarded by the context predicate `is-owner?` when they are accessed in the body of `show-private-agenda`. The `cals-array` variable is an array of the calendars that should be shown to the device owner. Agenda items are filtered using the `date-range` vari-

able before being added to the list of events for display.

The `show-public-agenda` procedure mode implements the context-dependent behaviour for showing public agenda items. It belongs to the `agenda` modal and is associated with the context predicate `is-not-owner?` that specifies the context it is constrained to run in. Additionally, the `show-public-agenda` mode is specified with the `default-config` that designates `suspend`, `restart`, and `immediate` as the interruption, resumption and state scoping strategies. The `restart` interruption strategy signifies that execution is restarted (cf. Section 4.8). The motivation for the `restart` strategy is that since the device may be given to several users, it is appropriate to restart the execution for each user. Similar to the `show-private-agenda` mode, accessing the `calendars` and `bg-colour` variable modals evaluate to the appropriate value for the context of use.

Evaluation

The previous section has presented the implementation of the *Kalenda* application in Flute. Below we highlight the benefits of using Flute to implement such an application.

- In the *Kalenda* implementation, we use Flute's `ctx-event` to represent the context source `current-user` as a reactive value. As `current-user` is a reactive value, context predicates `is-owner?` and `is-not-owner?` operate on it without explicit use of event handlers. Additionally, the developer does not need to worry about ensuring that its value is up-to-date. The Flute runtime ensures that the value of `current-user` automatically updated whenever a new value becomes available from the device orientation sensor (cf. **Requirement** *R*.1 *Chained Context Reactions*).

- The context-dependent behaviours of the agenda are implemented by the `show-private-agenda` and `show-public-agenda` procedure modes. Each agenda mode is associated with a context predicate that is not only used to determine its applicability but also for constraining its entire execution to the correct context. The Flute language runtime ensures that the context predicate is satisfied throughout the procedure mode's execution. This eliminates the need for explicit context checks in the body of the mode (cf. **Requirement** *R*.2 *Context-dependent interruptions*). Additionally, the support for modes of variables enable the creation of variables that evaluate to a different value depending on the current context of use.

- The modal abstraction enables grouping of related behavioural or variable modes under a single entity. New modes can be added at runtime as required. That is, new context-dependent behavioural modes can be added to the `agenda` modal without requiring modification to the `show-private-agenda` and `show-public-agenda` procedure modes. When the `agenda` modal is invoked, the procedure mode whose context predicate is satisfied is selected for execution. This eliminates the need to select manually the right mode to execute for the current context of use (cf. **Requirement** *R*.4 *Contextual dispatch*). Moreover, the dispatching process is automatically is repeated whenever a relevant context change is observed. Thus new modes that are added to a modal at runtime become part of the potential modes the dispatcher can select from (cf. **Requirement** *R*.5 *Reactive dispatch*).

- Each agenda mode is associated with an interruption strategy, a resumption strategy, and a state scoping strategy. For instance, the `show-private-agenda` mode is associated with (suspend resume isolated). This designates that the mode's execution is suspended if the context predicate is no longer satisfied, resumed if the context predicate becomes satisfied again, and that any state changes made during the mode's execution remain only *locally* visible. The developer does not need to worry about scoping the state changes that are performed during the execution of a mode (cf. **Requirements** *R*.2 *Context-dependent interruptions*, *R*.3 *Context-dependent resumptions*, and *R*.6 *Reactive scope management*).

8.5 Implementing the *Pulinta* Application

The second application that is included on the iFlute platform is the *Pulinta* application. As introduced in Section 8.2, the *Pulinta* is enhanced with context-awareness such that it is launched when the user enters a printer room. Additionally, the printing behaviour is enhanced with context-awareness such the printing of confidential documents continues only if the user is alone in the printer room. The implementation of the *Pulinta* application is structured as follows.

- The procedure modal of the *Pulinta* application is the `printing` modal. It groups together context-dependent behaviours (procedure modes) of the *Kalenda* application. The `printing` modal specifies the motion detector as the context source. In this scenario, we use the

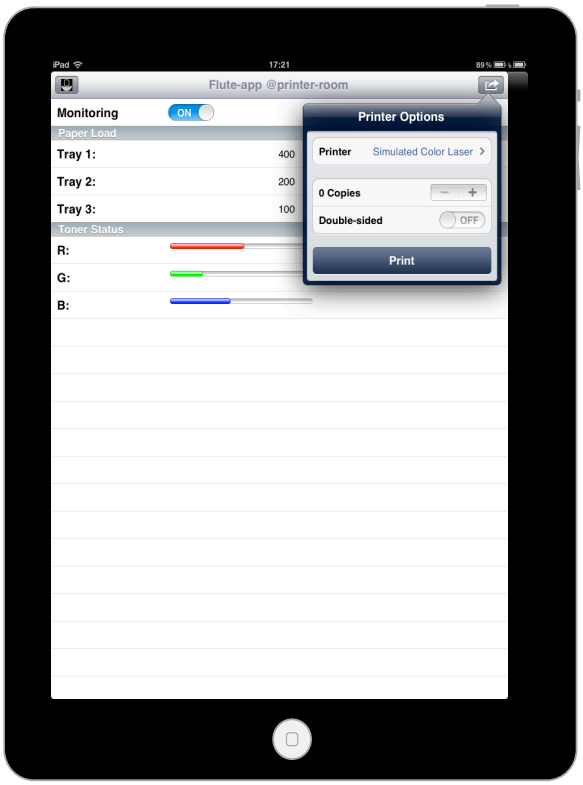

Figure 8.4: The *Pulinta* Application running when the user is in the printer room.

Modal	**Mode**	**Context predicate**	**Context source**
printing	confidential-printing	alone?	motion-detector
	regular-printing	people-nearby?	
documents	confidential documents	alone?	
	regular documents	people-nearby?	

Table 8.3: Modals and modes of the *Pulinta* application.

proximity sensor of the iOS device to indicate whether a user is alone or not.

- There are two procedure modes of the *Pulinta* application: the `confidential-printing` mode for printing confidential documents, and the `regular-printing` mode for non-confidential documents. The `confidential-printing` mode should only be executed when the user is alone in the printer room.

- In addition, the *Kalenda* application includes the `documents` variable modal that evaluates to the appropriate list of documents to print depending on the current context of use (i.e., the user is alone or not).

Listing 8.5 shows the implementation of the *Pulinta* application in Flute. For brevity the implementation eliminates the GUI concerns of the application (a complete record of the implementation is provided in the appendix).

Listing 8.5: Implementation of the *Pulinta* application in Flute.

```
1  ;defining the context source for user detection
2  (define motion-detector (ctx-event))
3
4  (define printing-assistant
5    (mode (flute-apps)
6      (printer-room? location)
7      (config)
8      (lambda ()
9       (define documents (modal (motion-detector)))
10
11      (define printing (modal (motion-detector)
12        ;shared variable for paper level
13        (define paper-level (tray-load))))
14
15      ...
16      (mode (documents)
17        (alone? motion-detector)
18        (filter confidential? app-directory))
19
20      (mode (documents)
21        (people-nearby? motion-detector)
22        (filter regular? app-directory))
23
24      (define confidential-printing
25        (mode (printing)
```

```
26          (alone? motion-detector)
27          (default-config)
28          (lambda ()
29            (define print-queue documents)
30            (define header "Confidential")
31            (define owner  user-name)
32            ;loop over documents
33            ;add metadata and print
34            (for-each
35              (lambda (doc)
36                (metadata header owner doc)
37                (print doc) )
38              print-queue) ) ) )
39
40        (define regular-printing
41          (lambda (printing)
42            (people-nearby? motion-detector)
43            (default-config)
44            (lambda ()
45              (define print-queue documents)
46              ;loop over documents and print
47              (for-each
48                (lambda (doc)
49                  (print doc) )
50                print-queue) ) ) )
51        ...
52        (render-toner-paper-status)
53        (printing) ) ) )
```

The `printing-assistant` mode belongs to the `flute-apps` modal (cf. Listing 8.3) and is specified with a context predicate `printer-room?` which implies that the application is launched only when the user enters a printer room. Line 9 creates the `documents` variable modal whose value is a list of either confidential or regular documents depending on the current context of use when it is accessed (i.e., confidential documents if the user is alone in the printer room and regular documents if there is another person in the printer room). The presence of another person in a printer room is derived from the context source `motion-detector`. Line 11 creates the `printing` procedure modal. It consists of the `confidential-printing` and `regular-printing` modes. A context predicate is associated with each mode to specify when it should be executed. For instance, the `confidential-printing` mode is associated with the `alone?` context predicate, which implies that the mode should be

executed only if the user is alone in the printer room. Therefore, suppose the `confidential-printing` mode is executing and another person walks into the printer room, the printing is suspended and resumed from where it left off, when the person walks out of range.

Evaluation

As with the *Kalenda* application, using Flute to implement the *Pulinta* application showcases the following strengths of the Flute programming language.

- Since the `motion-detector` context source as a reactive value, the `alone?` and `people-nearby?` context predicates can operate on it without using explicit callbacks (cf. **Requirement** *R*.1 *Chained context reactions*).
- A context predicate is associated with each printing mode, which is used by the Flute runtime to ensure that the entire mode executions is constrained to run only in the correct context (cf. **Requirement** *R*.2 *Context-dependent interruptions*).
- The `documents` variable modal groups together different modes of the variable, while the `printing` modal serves as a grouping entity for different printing modes (i.e., the `regular-printing` and `confidential-printing` modes). Flute's dispatching mechanism ensures that the right printing and documents modes are selected based on the context predicates that are satisfied (cf. **Requirement** *R*.4 *Contextual dispatch*). Moreover, the dispatching process to select the right printing mode to execute as other people enter or leave the printer room (cf. **Requirement** *R*.5 *Reactive dispatch*).
- As in the *Kalenda* application, each printing mode is associated with interruption, resumption, and state scoping strategies (cf. **Requirements** *R*.2 *Context-dependent interruptions*, *R*.3 *Context-dependent resumptions*, and *R*.6 *Reactive scope management*).

8.6 Implementing the *Tasiki* Application

The third example is the *Tasiki* application, which is a context-aware task assistant application. The application launches to assist the user to perform certain tasks (e.g., making phone calls, and processing emails). For instance, when the user is in a conference room at his/her workplace, the application checks if there are any co-workers contacts to call and offers to initiate a phone

call. Similarly, when the user is in his/her car the application automatically offers to initiate phone calls to the business contacts. The implementation of the *Tasiki* application is structured as follows.

Modal	Mode	Context predicate	Context source
to-call	call-from-car	in-car?	location
	call-from-conference-room	at-conference-room?	
contacts-list	business-contacts	in-car?	
	co-workers-contacts	at-conference-room?	

Table 8.4: Modals and modes of the *Tasiki* application.

- The `to-call` is a procedure modal that groups together the context-dependent behavioural variations of the *Tasiki* application. It relies on the context source is `location` whose value indicates whether the user is in his car or in a conference room at his/her workplace.
- Currently, there are two context-dependent behavioural variations of the `to-call` modal: the `call-from-car` procedure mode, and the `call-from-conference-room` procedure mode.
- The variable modal `contacts-list` is a reference to the contacts database on the phone. It has two modes: the `business-contacts` and `co-workers-contacts` each representing the contacts to call in the respective category.

Listing 8.6 shows the implementation of the *Tasiki* application in Flute.

Listing 8.6: Implementation of the *Tasiki* application in Flute

```
1  ;context predicate definitions
2  (define (at-conference-room? user-location)
3    (equal? user-location 'conference-room))
4
5  (define (in-car? user-location)
6    (equal? user-location 'car))
7
8  (define (at-conference-room-or-car? user-location)
9    (or (at-conference-room? user-location)
10       (in-car? user-location)))
11
12 (define task-assistant
13   (mode (flute-apps)
14     (at-conference-room-or-car? location)
15     (config)
```

```
16    (lambda ()
17
18      (define to-call (modal (location)))
19
20      (define contacts-list (modal (location)))
21
22      (mode (contacts-list)
23        (at-conference-room? location)
24        (co-workers-contacts))
25
26      (mode (contacts-list)
27        (in-car? location)
28        (biz-contacts))
29
30      (define call-from-car
31        (mode (to-call)
32          (in-car? location)
33          (config)
34          (lambda ()
35            (show "calling business contacts")
36            (if (null? contacts-list)
37                (show "business contacts list is empty")
38                (let ((contact (car contacts-list)))
39                  (set! contacts-list (cdr contacts-list))
40                  (show        (contact-name contact))
41                  (dial        (phone-number contact))
42                  (turn-on-phone-speaker)
43                  (connect-to-car-speakers)
44                  (if (dial-next-contact? (user-response))
45                      (to-call)))))))
46
47      (define call-from-conference-room
48        (mode (to-call)
49          (at-conference-room? location)
50          (config)
51          (lambda ()
52            (show "calling co-workers contacts")
53            (if (null? contacts-list)
54                (show "co-workers contacts list is empty")
55                (let ((contact (car contacts-list)))
56                  (set! contacts-list (cdr contacts-list))
57                  (show        (contact-name contact))
58                  (dial        (phone-number contact))
59                  (turn-on-phone-speaker)
60                  (connect-to-conference-equipment)
61                  (if (dial-next-contact? (user-response))
62                      (to-call)))))))
63      (to-call))))
```

In the above implementation, `task-assistant` is a mode that represents the entire *Tasiki* application and belongs to the `flute-apps` modal (cf. the implementation of the iFlute platform in Section 8.3). The `task-assistant` is specified with the context predicate `at-conference-room-or-car?` (cf. Line 14), which implies that the application should only be launched when the user is either at his/her workplace's conference room or in his/her car. The `task-assistant` mode includes the `call-from-conference-room` procedure mode (cf. Lines 47-62) and `call-from-car` procedure mode (cf. Lines 30-45) that are grouped together under the `to-call` procedure modal. The variable modal `contacts-list` (cf. Line 20) has two modes for the business contacts and co-workers contacts to call. Both the `to-call` procedure modal and the `contacts-list` variable modal rely on the `location` reactive value as the context source for the context predicates of their modes.

Each mode is guarded by a context predicate that constrains it to run *only* under a specific context. For instance, the `call-from-car` mode is guarded by the `in-car?` predicate which ensures that it is only executed when the user in his/her car. Observe that procedure modes in this example are implemented in a *recursive style*. Each procedure modes calls the `to-call` modal, which re-dispatches over the correct procedure mode to execute again. A mode's execution starts by showing a message on the phone's screen showing the kind of contacts the user is calling. A contact is retrieved from the list of contacts `contacts-list`. As `contacts-list` is a variable modal, accessing it yields a different contacts list depending on the context of use (i.e., business contacts or co-workers contacts). After retrieving the contact, we perform a mutation on the `contacts-list` variable modal in order to update it to the remaining contacts to call. *Note that mutating a variable modal only affects the contacts list of the variable mode that corresponds to the current context of use (cf. Variable modal assignment semantics Section 6.4.1).* Next we show the contact's name on the screen, dial the phone number and turn the phone's speaker. A connection is then established to the car's speakers or conference room phone equipment depending on whether the user is in his/her car or is in the conference room. After the call is finished, the user is prompted whether he/she wants to continue with the next contact. If the user chooses to continue, the `to-call` modal is called again, and the dispatching process is repeated again. If the user moves out of a certain context (e.g., out of the car), the procedure mode's execution is suspended. The execution is resumed to proceed with the next contact the next time the user moves back to the right context.

Evaluation

The above implementation of the *Tasiki* application in Flute showcases the following strengths:

- The above implementation benefits from Flute's support for contextual and reactive dispatch to determine the right procedure mode to execute for current context of use without use of explicit conditional checks. By invoking the `to-call` modal, the Flute language runtime selects the right mode to execute calling mode.
- Flute's support for variable modals and variable modes enables the variable reference `contacts-list` to yield different contacts list depending on the current context of use. Moreover, when variable modal is mutated, the language runtime ensures that only the variable mode that matches the current context is affected. This enables use of a single reference to the contacts list and eliminates the need for extra checks to select the right contacts list.
- As with the *Kalenda* and *Pulinta* application, each procedure mode is associated with a configuration `config` that specifies the interruption, resumption and state scoping strategies that enable procedure executions to be interrupted and resumed at any moment.

In the above sections, we have presented the implementation of the context-aware applications that are deployed on the iFlute platform. Figure 8.5 depicts an overview of the executions of the each of the applications. For brevity, the figure shows only the execution paths for procedure modal, procedure modes, variable modals, and variable modes. The execution paths for regular Scheme procedures are not depicted. The arrows from the `call-from-car` and `call-from-conference-room` procedure modes to the `to-call` procedure modal depict recursion.

8.7 Related Work Revisited

In Chapter 3, we reviewed the state of the art for programming languages and techniques for context-aware applications. In this section, we compare our approach with the state of the art.

8.7.1 Comparing Flute with First-class Continuations

Having demonstrated the use of the Flute language in implementing example *reactive* context-aware applications, it is appropriate to show how closely

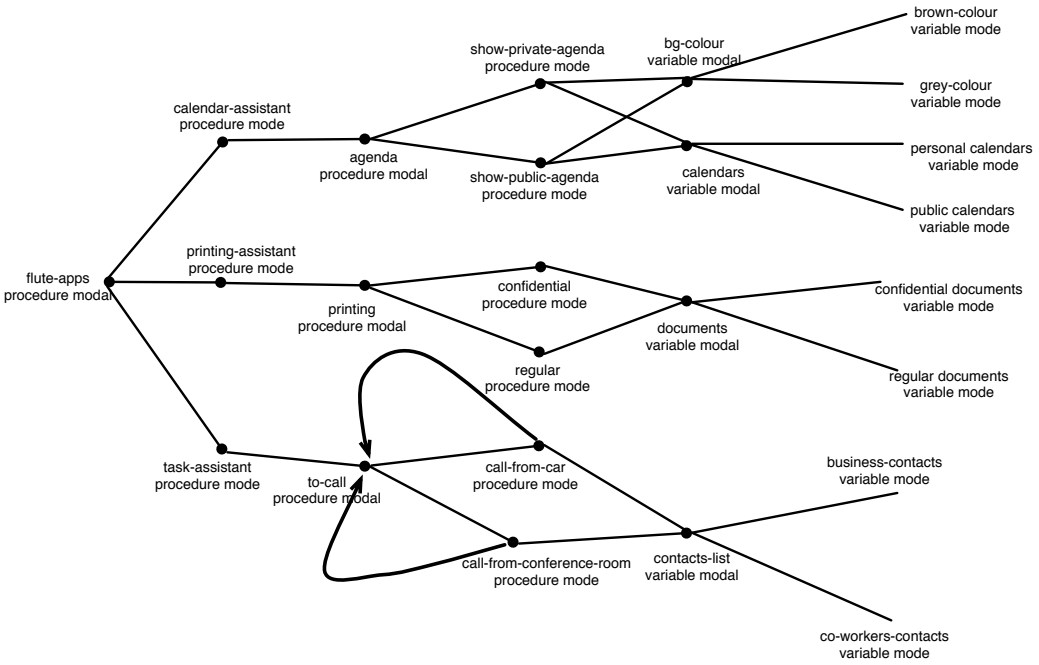

Figure 8.5: An overview of the execution paths for the context-aware applications on the iFlute platform.

related approaches compare with Flute when implementing similar applications. In this comparison, we consider parts of the *Tasiki* example and re-implement them using first-class continuations (cf. Section 3.4). We consider first-class continuations because from the survey of the state of the art (cf. Chapter 3), first-class continuations are one of the few approaches that enable interrupting and resuming ongoing procedure executions, albeit explicitly.

We re-implement the behaviour of the `call-from-car` procedure mode (cf. Listing 8.6) of the *Tasiki* application using `call/cc` construct of the Scheme programming language. In order to be able to express interruptions using first-class continuations, we will first implement a high-level construct `save/suspend` using `call/cc`.

Listing 8.7: Implementation of the `save/suspend` procedure using `call/cc`

```
1 (define escaper
2   (lambda ()
3     "escaper thunk"))
4
5 (define escaper-init
```

```
6     (lambda (continue)
7       (set! escaper continue)))
8
9   (call/cc escaper-init)
10
11  (define resume
12    (lambda ()
13      "no suspended execution found"))
14
15  (define suspend
16    (lambda (exit-procedure)
17      (escaper (exit-procedure))))
18
19  (define (save-execution execution)
20    (set! resume execution))
21
22  (define save/suspend
23    (lambda ()
24      (call/cc
25       (lambda (continue)
26         (save-execution continue)
27   (suspend (lambda () 'suspended))))))
```

Listing 8.7 shows the implementation of the `save/suspend` procedure (cf. Lines 22-27). The `save/suspend` procedure captures the current continuation using `call/cc` native procedure and suspends the current execution. We implement a number of helper procedures such as `escaper` in order to achieve the desired semantics. The captured continuation is saved in the `resume` variable.

Having implemented the `save/suspend` procedure, we will now show how it can be used to express interruptible context-dependent executions. Listing 8.8 shows the implementation of the `call-from-car` using the `save/suspend` procedure.

Listing 8.8: Implementation of the `call-from-car` mode using `call/cc`

```
1   (define call-from-car
2     (lambda (contacts-list)
3       (if (in-car? location)
4           (show "calling business contacts")
5           (save/suspend))
6       (if (null? contacts-list)
7           (show "the queue for business contacts is empty")
8           (let ((contact (car contacts-list)))
9             (if (in-car? location)
```

```
10              (show        (contact-name contact))
11              (save/suspend))
12          (if (in-car? location)
13              (dial        (phone-number contact))
14              (save/suspend))
15          (if (in-car? location)
16              (turn-on-phonespeaker)
17              (save/suspend))
18          (if (in-car? location)
19              (connect-to-car-speakers)
20              (save/suspend))
21          (if (in-car? location)
22              (if (dial-next-contact? (user-response))
23                  (call-from-car (cdr contacts-list)))
24              (save/suspend)))))
```

In order to ensure that the `call-from-car` procedure is constrained to execute *only* if the user is in the right context (i.e., in his/her car), the `in-car?` condition is inserted before every expression in the procedure body (Lines 3, 9, 12, 15, 18, and 21). If the context condition is *false* (i.e., if the user is not in his/her car), then the procedure execution is saved and suspended using the `save/suspend` procedure (Lines 5, 11, 14, 17, 20, and 24).

The above implementation of the `call-from-car` procedure is visibly convoluted because of the repetitive code that is needed for context checks, saving the execution state and suspending the execution. Clearly, implementing *reactive* context-aware applications in this style is difficult and error-prone. Even with all the context checks and manual execution state management, the above implementation is lacking since it does not include the logic of resuming suspended executions when previously unsatisfied condition later become satisfied again. Moreover, the developer is still required to perform manual dispatching to select the right procedure to execute for the current context of use. Flute's support for *contextual and reactive dispatch* eliminates the need for such explicit context checks and manual dispatch. It enables the developer to associate a context predicate with a procedure mode *only once* and the language runtime implicitly re-evaluates the context checks throughout the execution the procedure's body expressions. Moreover, Flute's support for reactive values facilitates event-driven resumption of suspended executions.

8.7.2 Discussion

To further showcase how Flute compares with the exisiting approaches, we present an overview of the evaluation of the state of the art alongside the

	Chained Context Reactions	Context-dependent Interruptions	Context-dependent Resumptions	Contextual Dispatch	Reactive Dispatch	Reactive Scope Management
Flute	✔reactive context sources	✔procedure modes and interruption strategies	✔resumption strategies	✔modals and modes	✔	✔state scoping strategies
Context-oriented Programming Languages						
ContextL and other Layer-based COP Languages	✖	✖	✖	✔	✖	✖
Lambic	✖	✖	✖	✔	✖	✖
Ambience	✖	✖	✖	✔	✖	✖
Contextual Values	✖	✖	✖	✔variables	✖	✔`scoped` construct
Functional Reactive Programming Languages						
FrTime and its Siblings	✔behaviours and events	✖	✖	✖	✔the same function re-evaluation	✖
Reactive SML	✖	✔explicit	✔explicit	✖	✖	✖
Advanced Control Flow Constructs						
Coroutines	✖	✔explicit	✔explicit	✖	✖	✖
First-class Continuations	✖	✔explicit	✔explicit	✖	✖	✖
Threads (Cooperative)	✖	✔explicit	✔explicit	✖	✖	✖
Other Programming Language Facilities						
Guards	✖	✔blocking at the beginning	✔from the beginning	✖	✖	✖
Assertions	✖	✔explicit abort	✖	✖	✖	✖
Invariants	✖	✔only at the beginning and the end	✖	✖	✖	✖

Table 8.5: Comparing Flute with existing programming language technologies for context-aware applications.

Flute language (cf. Table 8.5). As Table 8.5 shows the Flute language satisfies all the programming requirements for *reactive* context-aware applications (cf. Section 2.6).

- By representing context sources as reactive values Flute satisfies the requirement $R.1$ of *Chained context reactions*. The enables implementing context predicates as reactions to context changes without using callbacks. Moreover, Flute's language runtime automatically propagates context changes among context sources that depend on each other.

- By supporting procedure modes and variables, Flute satisfies the requirement $R.2$ of *Context-dependent interruptions*. The Flute language runtime ensures that the context predicates that are associated with modes are satisfied throughout the procedure execution. This eliminates the need for explicit context checks in the procedure body.

- Through the modal abstraction for grouping procedure modes or variables, Flute to satisfies the requirement $R.4$ of *Contextual dispatch*. The right mode to execute for the current context of use is selected based on the context predicate that is satisfied. Moreover, new modes can be added to a modal as requiring without requiring modifications to existing modes.

- Flute's language runtime support for continuous selection of appropriate modes to execute satisfies the requirement $R.5$ of *Reactive dispatch*. Flute ensures that modes that are added to a modal at runtime become potential candidates for the dispatcher to select from.

- Flute provides interruption and resumption strategies to enable the developer specify what to do when the context predicate is no longer satisfied or becomes satisfied again. By doing so Flute satisfies the requirements $R.2$ of *Context-dependent interruptions*, and $R.3$ of *Context-dependent resumptions*.

- Finally, Flute provides a number of state scoping strategies to enable the developer specify how to scope the state changes made during a mode's execution. By doing so Flute satisfies the requirement $R.6$ of *Reactive scope management*.

8.8 Chapter Summary

In this chapter, we have demonstrated the Flute language in action by using it to implement concrete examples of *reactive* context-aware applications. For each application, we evaluated the Flute language against the programming language requirements for *reactive* context-aware applications (cf. Section 2.6). We subsequently compared Flute with first-class continuations (which is a closely related approach that supports interruptions and resumptions of executions), by implementing one of the applications using the continuation abstraction of the Scheme programming language. Furthermore, we have presented an overview of the evaluation of existing approaches alongside the Flute language, which shows that Flute satisfies all the programming language requirements for *reactive* context-aware applications.

Chapter 9

Conclusions and Future Work

Contents

In this concluding chapter we give a summary of how the preceding chapters combine to answer the questions that we promised to tackle in Chapter 1 and the software engineering issues that we discussed in Chapter 2. We further discuss the limitations of our work and also provide insights into potential research directions of our work.

9.1 Restating the Problem Statement

> "*I'm talking about the limitations of programming which force the programmer to think like the computer rather than having the computer think more like the programmer*"–Dmitriev Sergey [Dmi04].

Current programming languages fall short of providing support for developing context-aware applications that must **react promptly to a sudden context change** – especially if such a context change occurs in the middle of an ongoing procedure execution. Consequently, developers have little choice but to resort to explicit management of the execution state (saving and restoring the procedure execution state between context changes) and explicit context checks (to ensure that the procedure execution is **always constrained to run only in the correct context**). However, the **unpredictable nature of context changes** renders it almost impossible for

the developer to know beforehand at which points in the procedure body to implement the above concerns. Doing this manually may result in incorrect application behaviour, such as a procedure continuing to run in a wrong context. Also, the developer must ensure manually that the execution environment remains in a consistent state between any interruptions caused by context changes.

Programming language requirements. To tackle the above problems, we put forward requirements that should be satisfied by a programming language designed for developing *reactive* context-aware applications. Table 9.1 summarises those programming language requirements.

	Language Requirement
$R.1$	Chained Context Reactions
$R.2$	Context-dependent Interruptions
$R.3$	Context-dependent Resumptions
$R.4$	Contextual Dispatch
$R.5$	Reactive Dispatch
$R.6$	Reactive Scope Management

Table 9.1: Revisiting the language requirements.

Survey of the state of the art. After putting forward the requirements that should be satisfied by a programming language for *reactive* context-aware applications, we performed a comprehensive study to evaluate the strengths and limitations of the existing approaches. We surveyed the state of the art of (context-oriented) programming languages and techniques that can be used to develop context-aware applications (cf. Chapter 3). The survey reveals that none of the reviewed approaches satisfies all the requirements. To address the above requirements, we proposed the *interruptible context-dependent executions* model and instantiated it in the Flute language. These research artefacts form the contributions of the research presented in this dissertation that we summarise in the next section.

9.2 Summary of the Contributions

The main goal of our research was to design and develop a programming language that facilitates the development of *reactive* context-aware applications. The programming language should satisfy the requirements summarised in

Table 9.1. We have achieved that goal and satisfied all the requirements through the following contributions. Our contributions are backed by an executable semantics that runs on a mobile platform.

9.2.1 The ICoDE Model

In Chapter 4, we proposed the interruptible context-dependent executions model (ICoDE) [BVDR+12]. The ICoDE model defines the properties and boundaries of a programming language designed for developing *reactive* context-aware applications. For each language property, we discussed design considerations that need to be taken into account in order to support it in a concrete programming language. Below we summarise the distillation of the properties of the interruptible context-dependent executions model.

Predicated procedures. We argued that each context-dependent procedure should be associated with a context predicate that is implicitly checked throughout the execution of the procedure body. This property satisfies the requirement of **Context-dependent interruptions** ($R.2$). Related predicated procedures are grouped together under a single identity. In addition, any variables that are shared among the predicated procedures belonging to the same identity can be specified as part of the group identity definition. New predicated procedures can be added to an existing grouping entity at runtime without requiring any modifications of the existing predicated procedures.

Representing context as reactive values. We argued that *reactive values* are suitable abstractions for representing context sources. This makes it possible to compose context predicates with the rest of a context-dependent program without having to use explicit event handlers. Moreover, representing context as reactive values facilitates the event-driven resumption of suspended executions. This property satisfies the requirement of **Chained context reactions** ($R.1$).

Reactive dispatching. We argued that the selection of the predicated procedure to run for the current context should be based on the context predicate that evaluates to true. By supporting this kind of dispatching mechanism the ICoDE model satisfies the requirement of **Contextual dispatch** ($R.4$). Moreover, the dispatching process is continuously repeated to take into account of any new context changes. This satisfies the requirement of **reactive dispatch** ($R.5$).

Interruptible executions. This property ensures that the execution of a predicated procedure is constrained to run only under the its prescribed

context predicate. If the context predicate is no longer satisfied the execution can be interrupted based on the developer-specified interruption strategies: *suspend* and *abort*. Additionally, the language should provide a construct that enables developers to demarcate certain critical regions in a program as "uninterruptible". This property satisfies the requirement of **Context-dependent interruptions** ($R.2$).

Resumable executions. This property ensures that the execution of a previously interrupted procedure execution can be later reinstated. It is desirable that an ICoDE language enables the developer to specify a resumption strategy: *resume* or *restart*. This property satisfies the requirement of **Context-dependent resumptions** ($R.3$).

Scoped state changes. Due to the fact that the execution of predicated procedures can be suspended or resumed at a later moment, it may result in situations where state changes performed during the execution of one procedure become visible to other executions. We identified three state scoping strategies, namely, *immediate visibility*, *deferred visibility*, and *isolated visibility* – that a programming language should provide to the developer to scope the visibility of state changes. This property satisfies the requirement of **Reactive scope management** ($R.6$).

9.2.2 The iScheme Mobile Language Laboratory

In Chapter 5, we presented *iScheme* [BVB+12], a language laboratory that we built to facilitate experimenting with the language constructs and features for a programming language designed for developing *reactive* context-aware applications. iScheme blends the rich programming properties of the Scheme language and a state-of-the-art mobile device that is equipped with context sensors to enable realistic experiments. For our experiments, we chose Apple's iOS devices that include the iPhone smartphone and the iPad tablet. In order to realise the iScheme language laboratory:

- We ported Scheme, which is a small but rich interpreted language, to the iOS platform; a mobile operating system for the iOS devices.
- We engineered a language symbiosis between Scheme and Objective-C [Koc09]. With this language symbiosis in place, iScheme provides developers with an event-driven programming model for accessing iOS's context sensors with higher-order procedures used as event handlers.
- We built distribution constructs specially tailored for distributed mobile computing environments. These distribution constructs are based

on the *ambient-oriented programming model* [Ded06] and have built-in support for peer-to-peer service discovery, asynchronous remote messaging, and failure handling. This enables distribution concerns (e.g., accessing remote contextual information) to be encapsulated in high-level constructs while relieving developers of the difficulties engendered by distribution.

9.2.3 An ICoDE Language: Flute

In Chapter 6, we presented *Flute* [BVDR+12], a proof-of-concept programming language that we built on top of iScheme to facilitate the development of *reactive* context-aware applications. The Flute language adheres to the ICoDE model. We described its language constructs by way of a running example (i.e., a context-aware calendar application). We further validated the language constructs in Chapter 8. Flute has been implemented as a meta-interpreter on top of iScheme [BVB+12]. This executable semantics was presented in Chapter 7. By incorporating the interruptible context-dependent executions model, Flute satisfies the language requirements for *reactive* context-aware applications and answers the research questions formulated in Section 1.4. Below we summarise the abstractions of the Flute language and their mapping to the language requirements and research questions.

- Flutes supports *variable modes* that facilitate the creation of variables whose value depends on context.

- Flute supports *procedure modes* that enable developers to express predicated procedures with a single context predicate that is implicitly checked throughout the mode execution. By supporting modes of procedures and variables, Flute satisfies the requirement of **Context-dependent interruptions** ($R.2$) and answers Research Question 1: *How to constrain a procedure execution to only occur under a particular context condition?*

- Flute supports the *modal* abstraction that enables developers to group together related modes and specify variables that are shared by those modes. The design of modals facilitates adding new modes at runtime. By providing the modal abstraction, Flute answers Research Question 5: *How can context-dependent behaviours be expressed in such away that new unanticipated behavioural variations can be added as required without requiring modification of existing behaviour definitions?*

- Flute supports the *reactive dispatching* mechanism that continuously takes into account new context changes in order to select the applicable modes for the current context of use. By supporting the reactive dispatching mechanism, Flute satisfies the requirements of **Contextual dispatch** ($R.4$) and **Reactive dispatch** ($R.5$), and answers Research Question 6: *How can the dispatching process of which behavioural variation to execute be scheduled take into account of the current and the future context?*

- Flutes supports interruptible and resumable executions. It provides interruption strategies (*suspend* and *abort*) that enable the developer to specify what to do with the execution when the associated context predicate is no longer satisfied. Additionally, Flute provides resumption strategies (*resume* and *restart*) that enable the developer to specify what to do with the suspended execution when its associated context predicate later becomes satisfied again. Through its interruption and resumption strategies, Flute satisfies the requirements of **Context-dependent interruptions** ($R.2$) and **Context-dependent resumptions** ($R.3$) and answers Research Question 2: *What should happen when a context change occurs in the middle of an ongoing procedure execution? Should the execution be interrupted and possibly be resumed later on?*

- Flute provides a number of state scoping strategies (*immediate, deferred*, and *isolated*) that enable the developer to control the visibility of state changes to the shared state. By providing state scoping strategies, Flute satisfies the requirement of **Reactive scope management** ($R.6$) and answers Research Question 3: *How to ensure that a procedure execution occurs in a consistent state even when the execution is interrupted before its completion?*

- Flute provides the `ctx-event` construct for representing context sources as reactive values. Reactive values employ a push-driven evaluation model for automatic propagation of context changes among dependent context sources. Moreover, *reactive values* facilitate the event-driven resumption of suspended executions. By representing context sources as reactive values, Flute satisfies the requirement of **Chained context reactions** ($R.1$) and answers Research Question 4: *How can context be represented in the underlying programming language such that it can be manipulated, reacted upon, and combined with other programs?*

9.2.4 The iFlute Platform

In Chapter 2, we presented a visionary mobile application model where mobile platforms are enhanced with context awareness to automatically launch the appropriate application without requiring the user to explicitly to tap an icon to launch the application. In order to ground our vision we introduced a scenario (dubbed *BainomuAppies* in Kampala). In the *BainomuAppies* scenario, buses and minibuses in Kampala are equipped with an onboard digital platform that runs a suite of applications. The currently running application as well as its behaviour depends on contextual information such as the geolocation of the bus, the proximity of other buses and certain stops, and the identity of the passengers that happen to be onboard the bus at a certain moment in time. From this scenario, we derived programming language requirements that should be satisfied by a language designed for developing *reactive* context-aware applications.

In Chapter 8, we presented a prototype implementation of a mobile platform called the *iFlute platform* that epitomises that mobile application model. The iFlute platform runs on iOS and is developed using the Flute language and iScheme. The iFlute platform contains a suite of mobile applications for different contexts. To alleviate the burden of manually selecting which application to run for the current task, the iFlute platform is enhanced with context-awareness to automatically present the user the appropriate behaviour for the task at hand. When there is a context change, the running application's behaviour is *promptly interrupted*. The application's *execution state* is automatically saved between interruptions and the application is able to resume from where it left off at a later moment when the user goes back to the previous context. We developed example mobile applications using Flute and deployed them on the iFlute platform. These include a context-aware calendar application, a context-aware printer assistant, and a context-aware task guide.

9.3 Limitations and Future Work

No research is without limitations and science will never be finished. The work presented in this dissertation is not an exception to that "rule". Below, we discuss some of the limitations of our work as well as future research directions of our work.

9.3.1 Custom Strategies

As discussed in Section 6.5.4, the interruption and resumption of a procedure execution is based on a developer-specified strategy. The instantiation of the ICoDE model in Flute offers the developer a number of strategies to select from when defining a mode. As discussed in Section 7.5.6 the Flute meta-interpreter is structured to allow the definition of new interruption and resumption strategies. However, Flute does not offer any meta-level hooks that enable the developer to create custom strategies without modifying the interpreter.

Such hooks would empower the developer with the flexibility to define different interruption and resumption strategies other than the predefined ones. Realising such support requires exposing the evaluation process to the Flute programs since the interruption and resumption strategies involve manipulating the evaluation process. A similar mechanism could be imagined to enable the developer to define custom scoping strategies to control the visibility of state change in an application-specific manner.

9.3.2 Garbage Collection of Suspended Executions

Garbage collection is an essential feature of modern programming languages and so it is for an ICoDE language. As already discussed in the previous chapters, procedure executions an ICoDE language are constrained to run only when their associated context predicates are satisfied. If the context predicate is no longer satisfied the procedure execution is interrupted (suspended or aborted). A question that arises is how long should a suspended procedure execution be stored before being subjected to garbage collection? In the best case scenario the suspended execution will be resumed at a later point when its associated context predicate becomes satisfied again. However, it is not always the case that every suspended procedure execution will be resumed at some point in the future. In the worst case scenario, the context predicate that is associated with a suspended execution *may never* become satisfied again. Such executions should be automatically garbage collected by the language runtime.

The current implementation of the Flute language does not embed any special techniques to identify such "never satisfied again" suspended executions. The main difficulty is due to the fact that it is almost impossible to a *priori* identify context predicates that will never be satisfied again. Potential solutions to this limitation include enabling the developer to specify a context condition-based *lease* such that when it expires and the corresponding execution that is still suspended is automatically garbage collected [BCV+09].

9.3.3 Evaluation Overhead

The interruptible context-dependent executions model advocates for interrupting a procedure execution at any moment during the execution. To this end, the associated context predicate is re-evaluated at *every* step of the evaluation. This incurs an extra evaluation overhead because of extra checks and evaluation of context predicates that needs to be performed. A possible solution to this limitation is to explore the use of interrupts (as in operating systems [MR97]). Instead of re-evaluating the context predicate at *every* evaluation step, the evaluation of a procedure body can continue evaluating the procedure body expressions until an interrupt signal is raised indicating that a context source has received a new value. Upon receiving the interrupt signal the evaluator can the re-evaluate the context predicate. This can reduce the overhead of the unnecessary re-evaluation of context predicates even when their context sources have not received new values. Also, while exploring the design spaces for the ICoDE model, we looked at the possibility of enabling the developer to explicitly demarcate certain regions of the procedure body that are "interruptible" instead of the current choice of enabling the developer demarcate the "uninterruptible" regions. While such an approach would limit the extra evaluation step to certain regions, it is not always possible to identify such "interrupted regions" as context changes can potentially occur at any moment as discussed in Section 2.3.

9.3.4 Ambiguous Context Predicates

Another limitation which is not specific to our approach but common to other predicate-based dispatch approaches is the *predicate ambiguity problem* [Mil04, Val11]. Ambiguous context predicates may arise either when there is no context predicate that is satisfied or there are multiple context predicates that are satisfied at the same time. In case there is no context predicate that is satisfied, this is not an issue for an ICoDE language because the dispatching process is implicitly scheduled to be repeated when the relevant context sources receive new values. Hence, a previously unsatisfied predicate may later become satisfied when the dispatching process is repeated in reaction to a context change. In case there are multiple context predicates that are satisfied at the same time there is a need to specify the mode to execute or the action to perform. The current instantiation of the ICoDE model in Flute relies on the developer to write mutually exclusive context predicates for modes that belong to the same modal. However, sometimes the developer may specify ambiguous context predicates where multiple predicates are satisfied at the same time. When the Flute dispatcher encounters

such cases, an *ambiguous context predicates* exception is raised. However, this is one design choice and there is room for exploring the solution space to this problem. Previous predicate dispatching approaches attempt to resolve ambiguities by verifying that all predicates are mutually exclusive and that at least one predicate must be *true*, at compile time. However, statically verifying the mutual exclusiveness of context predicates is only possible if the language for expressing context predicates is limited and also if context predicates are restricted to operate on a known domain of values. As the ICoDE model does not impose any restriction on the domain values for context predicates or the language for expressing context predicates, it is impossible to statistically verify the mutual exclusiveness of context predicates. A possible solution to this problem is to consider a language specified choice (e.g., selecting the first one) or a developer specified order (e.g., by assigning priorities to the context predicates).

9.3.5 Distributed Interruptible Executions

In this dissertation, we have explored the interruptible context-dependent executions (ICoDE) model in a non-distributed setting. A potential future research track is extending the ICoDE model in a distributed setting. Rather than context-dependent executions that are interrupted and resumed on the same device, executions can be spread across multiple devices. We can envision scenarios where it may be desirable to interrupt an ongoing context-dependent execution on one device and resume it on another device. In previous work [BVT+09], we investigated a *service partitioning* model to enable non-technical users to migrate certain parts of software applications from one device to another. It is therefore promising to investigate an integration of such service partitioning model with the ICoDE model. The resulting model could enable interrupting a running application and seamlessly resuming it on another device. Consider for instance, a user playing a game on a TV while at home. When the user leaves his/her home the running game is interrupted and migrated to the user's tablet device. The user can resume from the exact point the game was interrupted and continue playing the game on the tablet. A difficulty that we foresee lies in transmitting the execution state between devices. The support for distribution of the iScheme mobile language laboratory (cf. Chapter 5) provides the infrastructure for experimenting with distributed interruptible context-dependent executions.

9.3.6 An ICoDE Language for the Real World

In this dissertation, we have presented the first instantiation of the ICoDE model in the Flute programming language. In Chapter 7, we presented its proof-of-concept implementation through a meta-interpreter that is built on top of iScheme. A meta-interpreter is great for establishing the exact semantics of a programming language as well as experimenting with new language constructs and features. However, the major drawback of a meta-interpreter approach is that it introduces an evaluation overhead and suffers from performance issues. More research is needed on building a "real world" ICoDE programming language from the ground up using "a closer to the metal" programming language like C. Such an implementation approach would also permit building a *fully interruptible* system where every procedure call can be potentially interrupted. In the current instantiation of the ICoDE model in Flute, the evaluation of program expressions may result in procedure calls to iScheme or Objective-C methods. However, it is not possible to constrain the execution of the iScheme procedures or Objective-C methods to the context predicate of a procedure mode to which such expressions belong. Implementing an ICoDE programming language from the ground up would allow interrupting almost every expression in a procedure body. Another possible incarnation of the ICoDE model is to explore its integration with existing context-oriented programming approaches [HCN08, CH05, GMH07, VGC+10].

Another future research track is to apply an ICoDE language to bigger "real world" case studies of *reactive* context-aware applications. One of these is the *BainomuAppies* in Kampala scenario that was used to *motivate* reactive context-aware applications in Chapter 2. As mentioned in Chapter 8 that scenario was not feasible to implement within the scope of a Ph.D. dissertation because of its large scale and infrastructure demands (buses, minibuses, onboard computers, etc.). The benefits of using an ICoDE language to develop the *BainomuAppies* in Kampala scenario are twofold: our work will be further validated using a bigger case study (possibly revealing new research challenges), and the onboard digital platform on minibuses and buses in Kampala will have a socio-economic impact on the lives of the people in Kampala by providing them with an interactive digital information platform. Also, we believe that in the near future sensors that are available on mobile devices will become more and more precise (e.g., location sensors with a centimetre level precision [Hum12]). This will give rise to new scenarios of *reactive* context-aware applications that will make the need for the *interruptible context-dependent executions* model even more apparent.

Appendix A

Additional Details of iScheme

In this appendix we provide further details of the iScheme language laboratory. These include type conversions, interacting with native applications via iScheme and lessons learned.

A.1 Type conversions in iScheme

When Scheme values are passed to Objective-C methods, implicit conversion is performed to appropriate types in Objective-C (e.g., number to `NSNumber` and `string` to `NSString`). In addition, we provide procedures to perform explicit conversions of Scheme values to Objective-C values. For example, the `string->NSString` procedure converts a Scheme string to an Objective-C `NSString`. The `number->NSNumber` procedure converts a Scheme number to an Objective-C `NSNumber`. The `list->NSMArray` procedure converts a Scheme list to an Objective-C `NSMutableArray`.

By default, return values from Objective-C methods are wrapped as a generic `OBJC_TYPE` Scheme type, which is a Scheme value representation of Objective-C objects in the Scheme interpreter. iScheme provides procedures for converting Objective-C values to their Scheme counterparts. For example, the `NSString->string` procedure converts an Objective-C `NSString` to a Scheme string. The `NSNumber->number` procedure converts an Objective-C `NSNumber` to a Scheme number.

Let us illustrate the conversion procedures with an example of retrieving the device model. The `UIKit` framework provides the `UIDevice` Objective-C class that implements methods to access the device's information such as the name, the device model and the operating system name. We implement this example in Scheme using the symbiosis constructs as follows:

```
1 ; An example of retreiving a mode of an iOS device using Scheme
```

```
(define (show-my-device-details)
  (let* ((UIDevice        (OBJC-CLASS UIDevice))
         (device          (OBJC-SEND UIDevice currentDevice))
         (device-model    (OBJC-SEND device model))
         (model-string    (NSString->string device-model))
         (device-details (string-append "device model: " model-string)))
    (display device-details)))
```

The above code snippet shows the definition of the `show-my-device-details` procedure that makes use of the type conversion procedure `NSString->string`. The `(OBJC-CLASS UIDevice)` expression returns a reference to the `UIDevice` class that is then bound to a Scheme variable `UIDevice`. The `(OBJ-SEND UIDevice currentDevice)` expression invokes the class method `currentDevice` on the class `UIDevice` and returns the instance representing the current device. We first send the message `model` to retrieve the Objective-C's `NSString` object for the device model. We then use the construct `NSString->string` to convert the Objective-C string object to a Scheme string. Evaluating the expression `(show-my-device-details)` displays the model of the device as follows.

```
(show-my-device-details)
===> device model: iPad
```

A.2 Interacting with Native iOS Applications in iScheme

iScheme provides a scripting environment that enables developers to create Scheme programs that dynamically interact with the native iOS applications such as the music player, phone, SMS, calendar and contacts. This opens the way for developers to prototype new ideas and test them directly on the device without having to create a project, compile and deploy the application as is the case in the Objective-C development. For instance, one can easily develop a variation of the native iTunes application enriched with location information (e.g., to stop playing music when a user walks into a meeting room).

In the remainder of this section, we describe an example Scheme application that interacts with the iTunes native media player application on the iOS devices. Objective-C provides the *Media Player* framework that enables access to the media library and methods to play movies, music, audio podcasts,

and audio books. This framework enables developers to build applications that make use of the media facilities. Using the symbiosis constructs in Scheme, one can build such applications on the fly. The implementation of such an application in Scheme is shown below.

```
1  (define (iTunes-controller)
2    (let* ((Controller (OBJC-CLASS MPMusicPlayerController))
3           (MQuery     (OBJC-CLASS MPMediaQuery))
4           (query      (OBJC-SEND MQuery songsQuery))
5           (musicPlayer (OBJC-SEND Controller iPodMusicPlayer)))
6      (OBJC-SEND musicPlayer setQueueWithQuery: query)
7      (lambda (action)
8        (case action
9          ((play)       (OBJC-SEND musicPlayer play))
10         ((stop)       (OBJC-SEND musicPlayer stop))
11         ((pause)      (OBJC-SEND musicPlayer pause))
12         ((play-next)  (OBJC-SEND musicPlayer skipToNextItem)))))))
```

The `iTunes-controller` procedure reifies the behaviour of the media player application. The expression `(OBJC-CLASS MPMusicPlayerController)` loads the `MPMusicPlayerController` Objective-C class and binds it to the `Controller` variable. The `MPMusicPlayerController` class implements methods for retrieving the instance of the media player.

The `(OBJC-CLASS MPMediaQuery)` expression loads the `MPMediaQuery` class and binds it to the variable `Query`. The `MPMediaQuery` class implements methods for constructing media query types (such as albums, artists, or songs). In this example, we create a media query of the music items grouped and sorted by the song name, by invoking the method `songsQuery` on the `MPMediaQuery` class. Invoking the method `iPodMusicPlayer` on `MPMusicPlayerController` returns the reference to the device's iPod music player instance that is bound to the `musicPlayer` variable. Next, we set the playback queue by invoking the method `setQueueWithQuery:` on the music player with the query type. In this example, the playback queue contains all songs. In addition, the music player provides methods `play, pause, skipToNextItem` to control the playback queue.

The `iTunes-controller` procedure returns a dispatcher procedure [1] that takes one argument and performs the corresponding action (play, pause,

[1] iScheme provides a small prototype-based object system, which eliminates the need to write a dispatcher procedure. For didactical reasons we write all examples in a procedureal style.

stop, skip to next item) depending on the specified argument. For example, the following scripts can be evaluated directly on the device to play and forward to next media items.

```
1
2 ;; example usage
3 (define my-iTunes-controller (iTunes-controller))
4
5 ; to start playing
6 (my-iTunes-controller 'play)
7
8 ; to play next song
9 (my-iTunes-controller 'play-next)
```

A.3 Lessons Learned

In this section we put forward our experiences gathered from porting Scheme to the iPhone, implementing a language symbiosis between Objective-C and Scheme, and implementing constructs that ease the development of the event-driven applications for mobile devices. We generalise the key concepts to make our experience usable to port other programming languages to the iPhone device.

A.3.1 On Implementing Language Symbiosis with Objective-C

Implementing language symbiosis with Objective-C is possible because of its dynamism and reflective capabilities. The Objective-C runtime library provides procedures to perform introspection (e.g., access to the methods a class implements) and intercession (e.g., adding a class, replacing a method implementation) on Objective-C objects at runtime. Below, we summarise the language constructs that one needs to implement in order to realise an interaction with Objective-C. For each construct we point out the relevant key Objective-C runtime procedures required to implement it.

First, the symbiotic language needs to define a language construct for loading Objective-C classes. For this, the Objective-C runtime library provides the procedure `objc_getClass(const *class_name)` that takes the string name of a class and returns a pointer to the class definition. In our Scheme, we implemented the `OBJC-CLASS` construct.

Second, an construct is required to provide means to send messages to Objective-C instances from the symbiotic language.

To this end, the Objective-C runtime provides a procedure `objc_msgSend(theReceiver, theSelector, args)` that performs message sends to an Objective-C object (the receiver) given a name of a method (the selector), and an optional variable number of arguments. A string representing a method name can be converted to a selector using the `NSSelectorFromString` runtime library procedure. In our Scheme, we implemented the `OBJC-SEND` construct.

The third set of language constructs is type conversion procedures. Type conversion can be implicit – meaning that values of one language are automatically converted to another language as they cross the bridge, or explicit – meaning that the programmer makes use of the conversion procedures. In our implementation, we perform automatic conversion when Scheme values cross to Objective-C and provide the programmer with conversion procedures (such as `NSString->string`) to convert Objective-C objects to Scheme values.

Other than the language symbiosis constructs that enable interaction between two languages, there are extensions required in the symbiotic language. First, the symbiotic language needs to be extended with a generic representation of the Objective-C objects. For example, we extend the Scheme value types with the `OBJC_TYPE` as a wrapper for Objective-C objects in Scheme. Second, a native procedure needs to be added to the symbiotic language to handle the calls to Objective-C. For example, we extend our Scheme interpreter with the `SOC` native procedure that serves as an interface to the Objective-C world.

A.3.2 On the Method Call Overhead

Implementing language symbiosis between two languages involves a sacrifice on performance. For example, method calls from Scheme to Objective-C involve a significant overhead compared to method calls in plain Objective-C. We performed preliminary benchmarks to quantify the method call overhead caused by the symbiosis.

We measured the method call overhead (in ms) on the iPhone 3G with ARM1176 412MHz and 128MB RAM running iPhone OS 3.1.3. We considered three different methods that vary by the number of parameters (zero, one, and two). All three methods have empty bodies and the parameters are of type `NSNumber`. Table A.1 shows the times per method call for the three methods (in plain Objective-C versus calls from Scheme to Objective-C). For each measurement we performed one million method calls.

There is a significant increase in the times of the method calls from Scheme to Objective-C. We have not applied any optimisation techniques

Table A.1: Preliminary benchmarks on the method call overhead from Scheme to Objective-C.

		Number of parameters	
	0	1	2
Method calls in Objective-C (ms)	0.00022	0.00023	0.00025
Method calls from Scheme to Objective-C (ms)	0.13500	0.15500	0.16150

in our current implementation. One possible factor for this increase is the fact that every Objective-C instance passed to Scheme needs to be wrapped as a Scheme generic value `OBJC_TYPE`. In addition, for each Scheme value passed to an Objective-C method, the symbiosis layer needs to check and perform a type conversion to the appropriate Objective-C object. As future, we would like to try performance enhancing techniques (such as caching of the selectors and method implementations).

A.4 iScheme Editor for the iPad

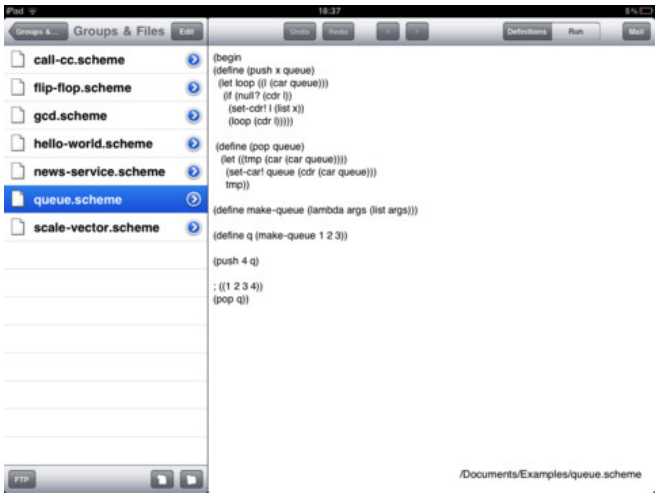

Figure A.1: The screen shot of the iScheme front-end editor running on the iPad device.

Appendix B

Flute Source Code

In this appendix we provide a complete record of the source code for the executable semantics of the Flute meta-interpreter.

B.1 Implementation for Reactive Values

```
1  ; Flute-FRP  implementation of FRP in iScheme.
2  ; Features:  automatic dependency management.
3  ; Context sources are represnted as reactive values.
4
5  ; (define <variable> (ctx-event <expression>))
6  ; (update-value! <context-source> <expression>)
7  ; (lift <native-function>)
8
9  (define event-tag 'event-tag)
10 (define tag-idx 0)
11 (define height-idx 1)
12 (define value-idx 2)
13 (define thunk-idx 3)
14 (define consumers-idx 4)
15 (define id-idx 5)
16 (define initial-height 0)
17 (define undefined 'undefined)
18
19 (define (new-event)
20   (define thunk (lambda () 'no-value))
21   (vector event-tag initial-height undefined thunk '() (getid)))
22
23 (define (ctx-event . value)
```

```
24    (let ((self (new-event)))
25      (if (and (pair? value) (event? (car value)))
26          (begin
27            (register self (car value))
28            (set-thunk!
29             self
30             (lambda ()
31               (update-value!
32                self
33                (current-value (car value)))))
34            (update-value! self (current-value (car value))))
35          (if (pair? value)
36              (update-value! self (car value))))
37      self))
38
39  (define (event? any)
40    (and (vector? any)
41         (= (vector-length any) (vector-length (new-event)))
42         (eq? (vector-ref any tag-idx) event-tag)))
43
44  (define (any-is-event? objects)
45    (ormap event? objects))
46
47  (define (event-values events)
48    (map current-value events))
49
50  (define (register consumer producer)
51    (if (not (consumer-exists? consumer producer))
52        (let* ((consumer-height (event-height consumer))
53               (producer-height (event-height producer))
54               (max-height (max consumer-height producer-height)))
55          (update-height! consumer (+ 1 max-height))
56          (new-consumer! producer consumer))))
57
58  (define (new-consumer! producer new-consumer)
59    (let ((consumers (vector-ref producer consumers-idx)))
60      (vector-set! producer consumers-idx (cons new-consumer consumers))))
61
62  (define (consumer-exists? consumer producer)
63    (member (event-id consumer)
64            (map event-id (event-consumers producer))))
65
66  (define (event-< event1 event2)
```

```
67    (< (pq-element-priority event1) (pq-element-priority event2)))
68
69  (define (alert-consumers consumers)
70    (let ((toupdate-pq (make-pq '() event-<)))
71      (for-each
72       (lambda (event)
73         (pq-insert! toupdate-pq event (event-height event)))
74       consumers)
75      (recompute (pq-sort toupdate-pq))))
76
77  (define (recompute stale-events)
78    (for-each
79     (lambda (event) ((event-thunk (cdr event))))
80     stale-events))
81
82  (define (update-value! event value)
83    (if (event? value)
84        (begin
85          (register event value)
86          (set-thunk!
87           event
88           (lambda ()
89             (update-value!
90              event
91              (current-value value))))))
92    (if (new-value? (current-value event) (current-value value))
93        (begin
94          (vector-set! event value-idx (current-value value))
95          (alert-consumers (event-consumers event)))))
96
97  (define (event-consumers event)
98    (vector-ref event consumers-idx))
99
100 (define (event-thunk event)
101   (vector-ref event thunk-idx))
102
103 (define (set-thunk! event thunk-value)
104   (vector-set! event thunk-idx thunk-value))
105
106 (define (current-value any)
107   (if (event? any)
108       (vector-ref any value-idx)
109       any))
```

```
110
111 (define (undefined? value)
112   (eq? value undefined))
113
114 (define (event-id event)
115   (vector-ref event id-idx))
116
117 (define (new-value? old-value new-value)
118   (or (undefined? new-value)
119       (not (eq? old-value new-value))))
120
121 (define (event-height event)
122   (vector-ref event height-idx))
123
124 (define (update-height! event height)
125   (vector-set! event height-idx height))
126
127 (define (frpify proc)
128   (lambda args
129     (let ((arguments (event-values args)))
130       (apply proc arguments))))
131
132 (define (lift proc)
133   (lambda args
134     (let* ((new-event (ctx-event))
135            (thunk
136             (lambda ()
137               (update-value!
138                new-event
139                (apply proc (event-values args))))))
140       (set-thunk! new-event thunk)
141       (thunk)
142       (for-each
143        (lambda (event)
144          (if (event? event)
145              (register new-event event)))
146        args)
147       new-event)))
```

B.2 The Flute Meta-Interpreter

As discussed in Section 7.5, the Flute meta-interpreter is implemented in a continuation-passing style (CPS) [FW08]. It explicitly passes a *continuation* parameter along with the environment. Structuring the interpreter in CPS is fundamental for realising Flute's semantics. In particular, CPS enables capturing and saving the execution context of an expression at any stage of the evaluation. The structure of the Flute meta-interpreter is based on the SLIP metacircular interpreter that is developed at the Software Languages Lab by Theo D'Hondt [D'H09].

```
1  ; Flute is an experimental programming language for developing
2  ; context-aware applications
3  ; The key idea behind Flute is that proc executions should be
4  ; interruptible and resumable at any moment.
5  ; This interpreter is built in a CPS style
6  ; The structure of the Flute meta-interpreter is based
7  ; on the SLIP metacircular interpreter by Theo D'Hondt
8  ; <expr>    ::= <computation>|<lambda>|<modal>|<mode>
9  ;                |<quote>|<variable>|
10 ; <modal>   ::= (modal   (<variable>+))
11 ; <modal>   ::= (modal   (<variable>+) <expr>+)
12 ; <mode>    ::= (mode    (<modal>) <expr> <expr>+)
13 ; <mode>    ::= (mode    (<modal>) <expr> (<config>) <lambda>)
14 ; <config>  ::= (create-config <expr>+)
15 ; <context-event>    ::= (ctx-event)
16 ; <context-event>    ::= (ctx-event <expr>)
17 ; <uninterruptible> ::= (continuous <expr>+)
18
19 (define meta-level-eval eval)
20
21 (define saved-executions (dictionary))
22 (define active-stms (make-stack))
23
24 (define beginning 'beginning)
25 (define during 'during)
26 (define interrupted 'interrupted)
27 (define immediate 'immediate)
28 (define deferred 'deferred)
29 (define isolated 'isolated)
30 (define resume 'resume)
31 (define suspend 'suspend)
32 (define restart 'restart)
```

```
33 (define abort 'abort)
34 (define completed 'completed)
35 (define aborted 'aborted)
36 (define no-true-pred 'no-true-pred)
37
38 ;Trascations operations
39 (define (make-transcation id)
40   (let ((transcation (dictionary)))
41     (put transcation 'ID id)
42     (put transcation 'read-log '())
43     (put transcation 'write-log '())
44     (put transcation 'commits '())
45     transcation))
46
47 (define (current-stm)
48   (if (stack-empty? active-stms)
49       (make-transcation (getid))
50       (top active-stms)))
51
52 (define (initialise-stm-log stm)
53   (push! active-stms stm))
54
55 (define (install-configs config)
56   (put config ':state-changes '(immediate))
57   (put config ':r-env   '(rt))
58   (put config ':p-true  '(restart '()))
59   (put config ':p-false '(suspend '())))
60
61
62 (define (current-configs config)
63   (config '?))
64
65 (define (create-config)
66   (let ((config (dictionary)))
67     (install-configs config)
68     config))
69
70 (define (p-false-config config)
71   (get config ':p-false))
72
73 (define (p-false-config! config strategy)
74   (del config ':p-false)
75   (put config ':p-false strategy)
```

```
76    config)
77
78  (define (p-true-config config)
79    (get config ':p-true))
80
81  (define (p-true-config! config strategy)
82    (del config ':p-true)
83    (put config ':p-true strategy)
84    config)
85
86  (define (state-changes-config config)
87    (get config ':state-changes))
88
89  (define (state-changes-config! config strategy)
90    (del config ':state-changes)
91    (put config ':state-changes strategy)
92    config)
93
94  (define (resumption-env-config config)
95    (get config ':r-env))
96
97  (define (resumption-env-config! config strategy)
98    (del config ':r-env)
99    (put config ':r-env strategy)
100   config)
101
102 (define modal-tag 'modal)
103 (define mode-tag 'mode)
104 (define variable-modal-tag 'variable-modal)
105 (define proc-modal-tag 'proc-modal)
106 (define modal-tag-idx 0)
107 (define modes-idx     1)
108 (define modal-envt-idx 2)
109 (define modal-event-sources-idx 3)
110 (define pred-idx       1)
111 (define mode-proc-idx  2)
112 (define mode-envt-idx  3)
113 (define (make-modal event-sources envt)
114   (let ((modes '()))
115     (vector modal-tag modes envt event-sources)))
116
117 (define (make-mode pred-expr mode-proc envt)
118   (vector mode-tag pred-expr mode-proc envt))
```

```
119
120 (define (modal-modes modal)
121   (vector-ref modal modes-idx))
122
123 (define (modal-modes! modal modes)
124   (vector-set! modal modes-idx modes))
125
126
127 (define (mode-pred mode)
128   (vector-ref mode pred-idx))
129
130 (define (modal-preds modal)
131   (define modes (modal-modes modal))
132   (map mode-pred modes))
133
134 (define  (modal-env modal)
135   (vector-ref modal modal-envt-idx))
136
137 (define (modal-event-sources modal)
138   (vector-ref modal modal-event-sources-idx))
139
140 (define (mode-env mode)
141   (vector-ref mode mode-envt-idx))
142
143 (define (mode-proc mode)
144   (vector-ref mode mode-proc-idx))
145
146 (define (modal? any)
147   (and (vector? any)
148        (eq? (vector-ref any modal-tag-idx) modal-tag)))
149
150 (define (add-mode! modal mode)
151   (let ((modes (vector-ref modal modes-idx)))
152     (vector-set! modal modes-idx (cons mode modes))))
153
154 (define (variable-modal? modal)
155   (and
156    (modal? modal)
157    (assoc variable-modal-tag modal)))
158
159 (define (proc-modal? modal)
160   (and
161    (modal? modal)
```

```
162     (assoc proc-modal-tag modal)))
163
164
165 (define (save-execution continue-point context-sources id)
166   (if (ormap event? context-sources)
167       (let ((execution-event (ctx-event)))
168         (let ((execution-thunk
169                (lambda ()
170                  (del saved-executions id)
171                  (del-thunk! execution-event)
172                  (continue-point))))
173           (set-thunk! execution-event execution-thunk)
174           (put saved-executions id execution-thunk))
175         (for-each
176          (lambda (context-source)
177            (if (event? context-source)
178                (register execution-event context-source)))
179          (listify context-sources)))))
180
181 (define (interruptible? seq)
182   (not (and (pair? seq)
183             (eq? (car seq) 'continuous))))
184
185 (define (dispatcher restart-dispatch event-sources)
186   (if (ormap event? event-sources)
187       (let* ((dispatch-event (ctx-event))
188              (dispatch-thunk
189               (lambda ()
190                 (restart-dispatch))))
191         (set-thunk! dispatch-event dispatch-thunk)
192
193         (for-each
194          (lambda (event-source)
195            (if (event? event-source)
196                (register dispatch-event event-source)))
197          (listify event-sources)))))
198
199       (define (wrap-native-proc native-proc)
200     (lambda (args continue env tailcall/unwrap)
201       (if (equal? tailcall/unwrap unwrap)
202           (continue  native-proc env)
203           (let ((native-value (apply native-proc args)))
204             (continue native-value env)))))
```

```
205
206
207 (define (unwrap-proc cps-proc continue env tailcall)
208   (define args 'args)
209   (cps-proc args continue env 'unwrap))
210
211 (define (cps-get expr continue env tailcall)
212   (define meta-get get)
213   (define dict (car expr))
214   (define key  (cadr expr))
215   (define (continue-after-unwrapping proc env)
216     (continue (meta-get proc key) env))
217   (unwrap-proc dict continue-after-unwrapping env unwrap))
218
219 (define (cps-nsstring->string expr continue env tailcall)
220   (define nsstring (car expr))
221   (define meta-nsstring->string NSString->string)
222   (continue (meta-nsstring->string nsstring) env))
223
224 (define (cps-string->nsstring expr continue env tailcall)
225   (define string (car expr))
226   (define meta-string->nsstring string->NSString)
227   (continue (meta-string->nsstring string) env))
228
229 (define (cps-list->nsmarray expr continue env tailcall)
230   (define lst (car expr))
231   (define meta-list->nsmarray list->NSMArray)
232   (continue (meta-list->nsmarray lst) env))
233
234 (define (cps-nsmarray->list expr continue env tailcall)
235   (define nsmarray (car expr))
236   (define meta-nsmarray->list NSMArray->list)
237   (continue (meta-nsmarray->list nsmarray) env))
238
239 (define (cps-update-value! expr continue env tailcall)
240   (define event (car expr))
241   (define new-event (cadr expr))
242   (define meta-update-value! update-value!)
243   (meta-update-value! event new-event)
244   (continue event env))
245
246 (define (cps-ctx-event expr continue env tailcall)
247   (define meta-ctx-event ctx-event)
```

```
248   (if (pair? expr)
249       (continue (meta-ctx-event (car expr)) env)
250       (continue (meta-ctx-event) env)))
251
252 (define (cps-apply expr continue env tailcall)
253   (define proc (car expr))
254   (define args (cadr expr))
255   (proc args continue env tailcall))
256
257 (define (cps-create-config expr continue env tailcall)
258   (define meta-create-config create-config)
259   (continue (meta-create-config) env))
260
261 (define (cps-current-configs expr continue env tailcall)
262   (define config (car expr))
263   (define meta-current-configs current-configs)
264   (continue (meta-current-configs config) env))
265
266 (define (cps-p-false-config! expr continue env tailcall)
267   (define config (car expr))
268   (define new-strategy (cadr expr))
269   (define meta-p-false-config! p-false-config!)
270   (continue (meta-p-false-config! config new-strategy) env))
271
272 (define (cps-p-true-config! expr continue env tailcall)
273   (define config (car expr))
274   (define new-strategy (cadr expr))
275   (define meta-true-config! p-true-config!)
276   (continue (meta-true-config! config new-strategy) env))
277
278 (define (cps-state-changes-config! expr continue env tailcall)
279   (define config (car expr))
280   (define new-strategy (cadr expr))
281   (define meta-state-changes-config! state-changes-config!)
282   (continue (meta-state-changes-config! config new-strategy) env))
283
284 (define (cps-resumption-env-config! expr continue env tailcall)
285   (define config (car expr))
286   (define new-strategy (cadr expr))
287   (define meta-resumption-env-config! resumption-env-config!)
288   (continue (meta-resumption-env-config! config new-strategy) env))
289
290
```

```
(define (cps-call-cc expr continue env tailcall)
  (define proc (car expr))
  (define (continuation args dynamic-continue dynamic-env tailcall)
    (continue (car args) env))
  (proc (list continuation) continue env tailcall))

(define (bind-variable variable value env)
  (define binding (cons variable value))
  (cons binding env))

(define (bind-params params args env)
  (if (symbol? params)
      (bind-variable params args env)
      (if (pair? params)
          (let*
              ((variable (car params))
               (value (car args ))
               (env (bind-variable variable value env)))
            (bind-params (cdr params) (cdr args) env))
          env)))


; <modal>       ::= (modal  (<variable>+))
; <modal>       ::= (modal  (<variable>+) <expr>+)
(define (eval-modal . exprs)
  (lambda (continue env tailcall)
    (define empty-envt '())
    (define (lookup variable)
      (define binding (assoc variable env))
      (if (pair? binding)
          (cdr binding)
          (meta-level-eval variable (interaction-env))))
    (if (pair? exprs)
        (let* ((params        (car exprs))
               (args          (map lookup params))
               (event-sources args)
               (modal-envt    (bind-params params args empty-envt)))
          (define (continue-after-modal-exprs value env-after-modal-exprs)
            (continue (make-modal event-sources env-after-modal-exprs) env))
          (if (pair? (cdr exprs))
              (eval-seq (cdr exprs)
                        continue-after-modal-exprs
                        modal-envt tailcall)
```

```
334                (continue-after-modal-exprs '()  modal-envt)))
335          (continue (make-modal '() empty-envt) env))))
336
337 ; <mode>         ::= (mode   (<modal>) <expr> <expr>+)
338 ; <mode>         ::= (mode   (<modal>) <expr> (<config>) <lambda>)
339 ; <config>       ::= (create-config <expr>+)
340 (define (eval-mode modal-expr pred-expr value-expr . exprs)
341   (lambda (continue env tailcall)
342     (define modal-binding (assoc (car modal-expr) env))
343
344     (define (continue-after-value-expr variable-value env-after-value-expr)
345       (define modal (cdr modal-binding))
346       (let* ((params      '())
347              (modal-envt (modal-env modal))
348              (mode-envt  (append modal-envt env))
349              (mode-proc  (make-proc params (list value-expr) mode-envt))
350              (mode       (make-mode pred-expr mode-proc mode-envt)))
351         (add-mode! modal mode)
352         (continue mode-proc env)))
353
354     (define (continue-after-config mode-config env-after-value-expr)
355       (define (mode-params exprs)
356         (car (cdr (car exprs))))
357       (define (mode-exprs exprs)
358         (cdr (cdr (car exprs))))
359       (define modal (cdr modal-binding))
360       (let* ((params         (mode-params exprs))
361              (body-exprs     (mode-exprs exprs))
362              (event-sources (modal-event-sources modal))
363              (modal-envt     (modal-env modal))
364              (mode-envt      (append modal-envt env))
365              (mode-proc      (make-c-proc params
366                                           (cons event-sources pred-expr)
367                                           mode-config body-exprs mode-envt))
368              (mode           (make-mode pred-expr mode-proc mode-envt)))
369         (add-mode! modal mode)
370         (continue mode-proc env)))
371
372     (if (pair? exprs)
373         (eval (car value-expr) continue-after-config  env #f)
374         (eval value-expr continue-after-value-expr  env #f))))
375
376
```

```
377 (define (eval-seq pred-expr config body-exprs args id
378                   proc-stm continue env tailcall)
379   (define head                  (car body-exprs))
380   (define tail                  (cdr body-exprs))
381   (define event-sources         (car pred-expr))
382   (define context-pred          (cdr pred-expr))
383   (define resumption-mechanism (car (p-true-config config)))
384   (define compensating-action  (cadr (p-false-config config)))
385   (define state-mechanism       (car (state-changes-config config)))
386   (define state-strategies      (dictionary))
387
388   (define (commit side-effects)
389     (let ((run (lambda (proc) (proc))))
390       (map run side-effects)))
391
392   (define (validate read-log env)
393     (if (not (null? read-log))
394         (let* ((variable  (caar read-log))
395                (log-value (cdr (car read-log)))
396                (binding   (assoc variable env))
397                (original-value     (cdr binding)))
398           (if (equal? log-value original-value)
399               (validate (cdr read-log) env)
400               #f))
401         #t))
402
403   (define (abort-execution)
404     (display "failed transcation validation !!"))
405
406   (define (isolated-strategy value env-after-seq)
407     (ignore-changes value))
408
409   (define (deferred-strategy value env-after-seq)
410     (if (equal? value completed)
411         (commit-changes value env-after-seq)
412         (ignore-changes value)))
413
414   (define (immediate-strategy value env-after-seq)
415     (commit-changes value env-after-seq))
416
417   (define (commit-changes value env-after-seq)
418     (let* ((active-stm (top active-stms))
419            (commits (get active-stm 'commits))
```

```
420             (read-log (get active-stm 'read-log)))
421         (if (or (equal? value completed) (equal? value interrupted))
422             (pop! active-stms))
423         (if (validate read-log env-after-seq)
424             (commit (reverse commits))
425             (abort-execution))))
426
427   (define (ignore-changes value)
428     (if (or (equal? value completed) (equal? value interrupted))
429         (pop! active-stms)
430         'done))
431
432   (define (install-state-strategies)
433     (put state-strategies immediate immediate-strategy)
434     (put state-strategies deferred deferred-strategy)
435     (put state-strategies isolated isolated-strategy)
436     'done)
437
438   (define (apply-state-strategy value env-after-seq)
439     (let ((state-strategy (get state-strategies state-mechanism)))
440       (if state-strategy
441           (state-strategy value env-after-seq))))
442
443   (define (continue-with-seq value env-after-seq)
444     (apply-state-strategy value env-after-seq)
445     (eval-seq pred-expr config tail args id
446           proc-stm continue env-after-seq tailcall))
447
448   (define (continue-after-context-pred boolean env-after-pred)
449     (define (continue-after-compensating-action
450           action-value env-after-user-action)
451       (let* ((state-strategy (get state-strategies state-mechanism)))
452         (if state-strategy
453             (state-strategy interrupted env-after-user-action)))
454       (continue interrupted env-after-user-action))
455     (if (and (eq? boolean #f) (interruptible? head))
456         (begin
457           (if (equal? resumption-mechanism resume)
458               (save-execution during resume-evaluation event-sources id))
459           (eval compensating-action continue-after-compensating-action
460             env-after-pred #f))
461         (if (null? tail)
462             (begin
```

```
463                 (apply-state-strategy completed env-after-pred)
464                 (eval head continue env-after-pred tailcall))
465               (eval head continue-with-seq env-after-pred #f))))
466
467     (define (resume-evaluation)
468       (if (equal? state-mechanism immediate)
469           (begin
470             (initialise-stm-log (make-transcation id)))
471           (initialise-stm-log proc-stm))
472       (eval (cdr pred-expr) continue-after-context-pred env #f))
473
474     (define (seq-evaluation-entry)
475       (install-state-strategies)
476       (eval context-pred continue-after-context-pred env #f))
477     (seq-evaluation-entry))
478
479   (define (make-c-proc params pred-expr config exprs env)
480     (lambda (args continue dynamic-env tailcall)
481       (define id (getid))
482       (define p-true-mechanisms  (p-true-config config))
483       (define lexical-env (bind-params params args env))
484
485       (define (continue-after-seq value env-after-seq)
486         (define (restart-strategy)
487           (define event-sources (car pred-expr))
488           (if (equal? value interrupted)
489               (save-execution during evaluation-entry event-sources id)
490               (save-execution during evaluation-entry event-sources id)))
491         (let* ((resumption-mechanism (car p-true-mechanisms))
492                (resumption-compensating-action (cadr p-true-mechanisms)))
493           (if (equal? resumption-mechanism restart)
494               (restart-strategy))
495           (continue value dynamic-env)))
496
497       (define (evaluation-entry)
498         (let* ((proc-stm (make-transcation id))
499                (lexical-env-after-pred
500                 (bind-variable 'pred pred-expr lexical-env))
501                (lexical-env
502                 (bind-variable 'default-config config lexical-env-after-pred)))
503           (initialise-stm-log proc-stm)
504           (eval-c-seq pred-expr config exprs args id
505                       proc-stm continue-after-seq lexical-env #t)))
```

```
506       (evaluation-entry)))
507
508   (define (eval-modal-dispatch modal continue env)
509     (define modes       (modal-modes modal))
510     (define preds       (modal-preds modal))
511     (define true-count-end   1)
512     (define true-count-start 0)
513
514     (define (iterate preds modes true-count mode-proc modes-after-dispatch)
515       (if (null? preds)
516           (if (= true-count true-count-end)
517               (begin
518                 (modal-modes! modal modes-after-dispatch)
519                 (continue mode-proc env))
520               (if (< true-count true-count-end)
521                   (continue #f env)
522                   (error "Ambiguous predicates for procedure modal" true-count)))
523           (let* ((head-pred       (car preds))
524                  (tail-preds      (cdr preds))
525                  (head-mode       (car modes))
526                  (tail-modes      (cdr modes))
527                  (mode-envt       (mode-env head-mode))
528                  (remaining-modes (cons head-mode modes-after-dispatch))
529                  (head-mode-proc  (mode-proc head-mode)))
530             (define (continue-after-pred value env)
531               (if value
532                   (iterate tail-preds  tail-modes (+ true-count 1)
533                            head-mode-proc modes-after-dispatch)
534                   (iterate tail-preds  tail-modes true-count
535                            mode-proc remaining-modes)))
536             (eval head-pred continue-after-pred  mode-envt #f))))
537     (iterate preds modes true-count-start #f '()))
538
539   (define (needs-result-handling? arg)
540     (or (equal? arg no-true-pred)
541         (equal? arg interrupted)))
542
543   (define (eval-application operator)
544     (lambda operands
545       (lambda (continue env tailcall)
546         (define binding (assoc operator env))
547
548         (define (continue-after-operator proc env-after-operator)
```

```
549           (define (eval-operands operands args env)
550             (define (continue-with-operands value env-with-operands)
551               (eval-operands (cdr operands) (cons value args)
552                              env-with-operands))
553             (if (null? operands)
554                 (proc (reverse args) continue env tailcall)
555                 (eval (car operands) continue-with-operands env #f)))
556           (if (equal? proc #f)
557               (continue no-true-pred env)
558               (eval-operands operands '() env-after-operator)))
559
560         (if binding
561             (let ((value (cdr binding)))
562               (if (modal? value)
563                   (let* ((modal value)
564                          (event-sources (modal-event-sources modal)))
565                     (define (start/restart-dispatch)
566                       (eval-modal-dispatch modal continue-after-operator env))
567
568                     (dispatcher start/restart-dispatch event-sources)
569                     (start/restart-dispatch))
570                   (eval operator continue-after-operator env #f)))
571               (eval operator continue-after-operator env #f)))))
572
573
574 (define (normalise-pred-exprs exprs env)
575   (let* ((event-sources '())
576          (lexical-pred (assoc 'pred env)))
577     (if (null? exprs)
578         (if lexical-pred
579             (cons event-sources (cddr lexical-pred))
580             (cons event-sources exprs))
581         (cons event-sources exprs))))
582
583 (define (normalise-pattern pattern)
584   (let ((normalised-pattern '(() () ())))
585     (if (null? pattern)
586         normalised-pattern
587         (if (null? (cdr pattern))
588             (begin
589               (set-car! normalised-pattern (car pattern))
590               normalised-pattern)
591             (if (null? (cddr pattern))
```

```
592                 (let* ((pred (cadr pattern))
593                        (pred-config (list pred '())))
594                   (set-cdr! pattern pred-config)
595                   pattern)
596                 pattern)))))
597
598 (define (normalise-config config env)
599   (let ((lexical-config (assoc 'default-config env)))
600     (if (null? config)
601         (list (cdr lexical-config))
602         config)))
603
604 (define (eval-if pred consequent . alternative)
605   (lambda (continue env tailcall)
606     (define (continue-after-pred boolean env-after-pred)
607       (if (eq? boolean #f)
608           (if (null? alternative)
609               (continue '() env-after-pred)
610               (eval (car alternative) continue env-after-pred tailcall))
611           (eval consequent continue env-after-pred tailcall)))
612     (eval pred continue-after-pred env #f)))
613
614 (define (eval-quote expr)
615   (lambda (continue env tailcall)
616     (continue expr env)))
617
618 (define (eval-seq exprs continue env tailcall)
619   (define head (car exprs))
620   (define tail (cdr exprs))
621   (define (continue-with-seq value env)
622     (eval-seq tail continue env tailcall))
623
624   (if (null? tail)
625       (eval head continue env tailcall)
626       (eval head continue-with-seq env #f)))
627
628 (define (make-proc params exprs env)
629   (lambda (args continue dynamic-env tailcall)
630     (define (continue-after-seq value env-after-seq)
631       (continue value dynamic-env))
632     (define lexical-env (bind-params params args env))
633     (if tailcall
634         (eval-seq exprs continue lexical-env #t)
```

```
635         (eval-seq exprs continue-after-seq lexical-env #t))))
636
637
638 (define (eval-define pattern . exprs)
639   (lambda (continue env tailcall)
640     (if (symbol? pattern)
641         (let* ((binding (cons pattern '()))
642                (env (cons binding env)))
643           (define (continue-after-expr value env-after-expr)
644             (set-cdr! binding value)
645             (continue value  env-after-expr))
646           (eval (car exprs) continue-after-expr env #f))
647         (let* ((binding (cons (car pattern) '()))
648                (env (cons binding env))
649                (proc (make-proc (cdr pattern) exprs env)))
650           (set-cdr! binding proc)
651           (continue proc env)))))
652
653 (define (eval-lambda params . exprs)
654   (lambda (continue env tailcall)
655     (continue (make-proc params exprs env) env)))
656
657 (define (evaluate-set! variable expression)
658   (lambda (continue env tailcall)
659     (define (continue-after-expression value env-after-expression)
660       (define binding (assoc variable env-after-expression))
661       (if binding
662           (if (modal? (cdr binding))
663               (mutate-modal-variable (cdr binding) value continue env)
664               (set-cdr! binding value))
665           (error "inaccessible variable: " variable))
666       (continue value env-after-expression))
667     (evaluate expression continue-after-expression env #f)))
668
669 (define (eval-c-set! variable expr)
670   (lambda (continue env tailcall)
671     (define (continue-after-expr value env-after-expr)
672       (let* ((binding (assoc variable env-after-expr))
673              (active-stm (current-stm))
674              (write-log (get active-stm 'write-log))
675              (read-log (get active-stm 'read-log))
676              (binding-write (assoc variable write-log))
677              (binding-read (assoc variable read-log))
```

```
678               (commits (get active-stm 'commits)))
679
680           (define (delay-side-effects)
681             (let ((todo (lambda () (set-cdr! binding value))))
682               (del active-stm 'commits)
683               (put active-stm 'commits (cons todo commits))))
684
685           (define (extend-read-log)
686             (let ((new-read-log (cons binding read-log)))
687               (del active-stm 'read-log)
688               (put active-stm 'read-log new-read-log)))
689
690           (define (extend-write-log)
691             (let* ((log-binding (cons variable value))
692                    (new-write-log (cons log-binding write-log)))
693               (del active-stm 'write-log)
694               (put active-stm 'write-log new-write-log)))
695
696           (if binding-write
697               (set-cdr! binding-write value)
698               (if binding-read
699                   (extend-write-log)
700                   (if binding
701                       (begin
702                         (extend-read-log)
703                         (extend-write-log))
704                       (error "inaccessible variable: " variable))))
705           (delay-side-effects)
706           (continue value env-after-expr)))
707       (eval expr continue-after-expr env #f)))
708
709
710   (define (eval-modal-variable modal continue env)
711     (define modes       (modal-modes modal))
712     (define preds       (modal-preds modal))
713     (define end-true-count   1)
714     (define start-true-count 0)
715     (define (iterate preds modes true-count mode-proc)
716       (if (null? preds)
717           (if (= true-count end-true-count)
718               (mode-proc '() continue env #f)
719               (error "Exactly one true pred is expected " true-count))
720           (let* ((head-pred       (car preds))
```

```
721                 (tail-preds      (cdr preds))
722                 (head-mode       (car modes))
723                 (tail-modes      (cdr modes))
724                 (mode-envt       (mode-env head-mode)))
725             (define (continue-after-pred value env)
726               (if value
727                   (iterate tail-preds  tail-modes (+ true-count 1)
728                            (mode-proc head-mode))
729                   (iterate tail-preds  tail-modes true-count mode-proc)))
730             (eval head-pred continue-after-pred mode-envt #f))))
731   (iterate preds modes start-true-count #f))
732
733 (define (eval-variable variable continue env)
734   (define binding (assoc variable env))
735   (if binding
736       (let ((value (cdr binding)))
737         (if (modal? value)
738             (eval-modal-variable value continue env)
739             (continue (cdr binding) env)))
740       (let ((native-value (meta-level-eval variable (interaction-env))))
741         (if (proc? native-value)
742             (continue (wrap-native-proc (frpify native-value)) env)
743             (continue native-value env)))))
744
745 (define (eval-c-ref variable)
746   (lambda (continue env tailcall)
747     (let* ((binding (assoc variable env))
748            (active-stm (current-stm))
749            (write-log (get active-stm 'write-log))
750            (read-log (get active-stm 'read-log))
751            (binding-write (assoc variable write-log))
752            (binding-read (assoc variable read-log))
753            (log-binding (cons variable 'nil)))
754       (define (extend-read-log)
755         (let ((new-read-log (cons binding read-log)))
756           (del active-stm 'read-log)
757           (put active-stm 'read-log new-read-log)))
758
759       (if binding-write
760           (continue (cdr binding-write) env)
761           (if binding
762               (begin
763                 (extend-read-log)
```

```
764                 (continue (cdr binding) env))
765               (let ((native-value (meta-level-eval variable (interaction-env))))
766                 (if (proc? native-value)
767                     (continue (wrap-native-proc (frpify native-value)) env)
768                     (continue native-value env))))))))
769
770 (define (eval-continuous . exprs)
771   (lambda (continue env tailcall)
772     (let* ((params '())
773            (args  '())
774            (proc (make-proc params exprs env)))
775       (eval (cons proc args) continue env #f))))
776
777
778 (define (eval expr continue env tailcall)
779   (cond
780     ((symbol? expr)
781      (eval-variable expr continue env))
782     ((pair? expr)
783      (let ((operator (car expr))
784            (operands (cdr expr)))
785        ((apply
786          (case operator
787            ((begin)              eval-begin        )
788            ((define)             eval-define       )
789            ((if)                 eval-if           )
790            ((lambda)             eval-lambda       )
791            ((quote)              eval-quote        )
792            ((set!)               eval-set!         )
793            ((let)                eval-let          )
794            ((let*)               eval-let*         )
795            ((continuous)         eval-continuous   )
796            ((modal)              eval-modal        )
797            ((mode)               eval-mode         )
798            (else                 (eval-application operator)))
799          operands) continue env tailcall)))
800     (else (continue expr env))))
801
802 (define natives
803   (list (cons 'apply cps-apply    )
804         (cons 'call-cc  cps-call-cc )
805         (cons 'update-value! cps-update-value!)
806         (cons 'ctx-event cps-ctx-event)
```

```
807         (cons 'create-config cps-create-config)
808         (cons 'p-false-config! cps-p-false-config!)
809         (cons 'p-true-config! cps-p-true-config!)
810         (cons 'state-changes-config! cps-state-changes-config!)
811         (cons 'current-configs cps-current-configs)
812         (cons 'default-config (create-config))
813         (cons 'NSString->string cps-nsstring->string)
814         (cons 'string->NSString cps-string->nsstring)
815         (cons 'list->NSMArray   cps-list->nsmarray)
816         (cons 'NSMArray->list   cps-nsmarray->list)))
```

Bibliography

[ADB+99] Gregory D. Abowd, Anind K. Dey, Peter J. Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a better understanding of context and context-awareness. In *Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*, HUC '99, pages 304–307, London, UK, UK, 1999. Springer-Verlag.

[AH00] Kenneth Anderson and Timothy J. Hickey. SILK – a playful blend of Scheme and Java. In *Proceedings of the Scheme and Functional Programming Workshop*, September 2000.

[App09] Apple Inc. Using the Java bridge, September 2009. http://developer.apple.com/legacy/mac/library/.

[App12] Malte Appeltauer. *Extending Context-oriented Programming to New Application Domains: Run-time Adaptation Support for Java*. PhD thesis, Hasso-Plattner-Institute, 2012.

[AS96] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996.

[BCC+11] Engineer Bainomugisha, Alfredo Càdiz, Pascal Costanza, Wolfgang De Meuter, Sebastiàn Gonzàlez, Kim Mens, Jorge Vallejos, and Tom Van Cutsem. Language engineering for mobile software. In Paulo Alencar Donald Cowan, editor, *Handbook of Research on Mobile Software Engineering: Design, Implementation and Emergent Applications*. IGI Global, 2011. Chapter.

[BCV+09] Elisa Gonzalez Boix, Tom Van Cutsem, Jorge Vallejos, Wolfgang De Meuter, and Theo D'Hondt. A leasing model to deal with partial failures in mobile ad hoc networks. In *TOOLS (47)*, pages 231–251, 2009.

[BDG+88] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common lisp object system specification. *SIGPLAN Not.*, 23(SI):1–142, September 1988.

[BDMD09] Engineer Bainomugisha, Wolfgang De Meuter, and Theo D'Hondt. Towards context-aware propagators: language constructs for context-aware adaptation dependencies. In *International Workshop on Context-Oriented Programming*, COP '09, pages 8:1–8:4, New York, NY, USA, 2009. ACM.

[BDR07] Matthias Baldauf, Schahram Dustdar, and Florian Rosenberg. A survey on context-aware systems. *International Journal Ad Hoc Ubiquitous Computing*, 2:263–277, June 2007.

[BDR09] Johan Brichau and Coen De Roover. Language-shifting objects from Java to Smalltalk: an exploration using JavaConnect. In *Proceedings of the International Workshop on Smalltalk Technologies*, IWST '09, pages 120–125, New York, NY, USA, 2009. ACM.

[BEM07] Ralph-Johan Back, Johannes Eriksson, and Magnus Myreen. Testing and verifying invariant based programs in the socos environment. In *Proceedings of the 1st international conference on Tests and proofs*, TAP'07, pages 61–78, Berlin, Heidelberg, 2007. Springer-Verlag.

[BLV+12] Engineer Bainomugisha, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. A survey on reactive programming. *ACM Computing Surveys*, 2012. (To appear).

[BM12] Engineer Bainomugisha and Wolfgang De Meuter. Open reactive dispatch. In *Proceedings of the 3rd international workshop on Free composition, co-located with SPLASH 2012*, FREECO-SPLASH 2012, New York, NY, USA, 2012. ACM.

[Bot98] Per Bothner. Kawa: compiling dynamic languages to the Java VM. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '98, pages 41–41, Berkeley, CA, USA, 1998. USENIX Association.

[BPV+12] Engineer Bainomugisha, Koosha Paridel, Jorge Vallejos, Yolande Berbers, and Wolfgang De Meuter. Flexub: Dynamic

subscriptions for publish/subscribe systems in manets. In *Distributed Applications and Interoperable Systems*, volume 7272 of *Lecture Notes in Computer Science*, pages 132–139. Springer Berlin / Heidelberg, 2012.

[BVB+12] Engineer Bainomugisha, Jorge Vallejos, Elisa Gonzalez Boix, Pascal Costanza, Theo D'Hondt, and Wolfgang De Meuter. Bringing Scheme programming to the iPhone - Experience. *Software, Practice Experience.*, 42(3):331–356, 2012.

[BVDR+12] 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 *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*, Onward! '12, pages 67–84, New York, NY, USA, 2012. ACM.

[BVT+09] Engineer Bainomugisha, Jorge Vallejos, Éric Tanter, Elisa Gonzalez Boix, Pascal Costanza, Wolfgang De Meuter, and Theo D'Hondt. Resilient actors: a runtime partitioning model for pervasive computing services. In *Proceedings of the 2009 international conference on Pervasive services*, ICPS '09, pages 31–40, New York, NY, USA, 2009. ACM.

[CEM03] Licia Capra, Wolfgang Emmerich, and Cecilia Mascolo. Carisma: Context-aware reflective middleware system for mobile applications. *IEEE Trans. Softw. Eng.*, 29(10):929–945, October 2003.

[CH05] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming: an overview of contextl. In *Proceedings of the 2005 symposium on Dynamic languages*, DLS '05, pages 1–10, New York, NY, USA, 2005. ACM.

[CHDM06] Pascal Costanza, Robert Hirschfeld, and Wolfgang De Meuter. Efficient layer activation for switching context-dependent behavior. In *Proceedings of the 7th joint conference on Modular Programming Languages*, JMLC'06, pages 84–103, Berlin, Heidelberg, 2006. Springer-Verlag.

[CK06] Gregory H. Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *European Symposium on Programming*, pages 294–308, 2006.

[CMB+07] Tom Van Cutsem, Stijn Mostinckx, Elisa Gonzalez Boix, Jessie Dedecker, and Wolfgang De Meuter. Ambienttalk: Object-oriented event-driven programming in mobile ad hoc networks. In *SCCC '07: Proceedings of the XXVI International Conference of the Chilean Society of Computer Science*, pages 3–12, Washington, DC, USA, 2007. IEEE Computer Society.

[Con63] Melvin E. Conway. Design of a separable transition-diagram compiler. *Communications of The ACM*, 6:396–408, 1963.

[Cos10] Pascal Costanza. Contextscheme, 2010. http://www.p-cos.net/context-scheme.html.

[DAS01] Anind K. Dey, Gregory D. Abowd, and Daniel Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Hum.-Comput. Interact.*, 16(2):97–166, December 2001.

[De 04] Wolfgang De Meuter. *Move Considered Harmful: A Language Design Approach to Mobility and Distribution for Open Networks*. PhD thesis, Vrije Universiteit Brussel, 2004.

[Ded06] Jessie Dedecker. *Ambient-oriented Programming*. PhD thesis, Vrije Universiteit Brussel, 2006.

[D'H09] Theo D'Hondt. Slip: a simple language implementation platform, 2009. http://soft.vub.ac.be/~tjdhondt/PLE.

[D'H10] Theo D'Hondt. The skem interpreter, 2010. http://soft.vub.ac.be/soft/skem.

[Dmi04] Sergey Dmitriev. Language oriented programming: The next programming paradigm. *JetBrains onBoard*, 1(2), 2004.

[Edw09] Jonathan Edwards. Coherent reaction. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 925–932, New York, NY, USA, 2009. ACM.

[EFGK03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.

[Epp11] Sarah Rotman Epps. What the post-pc era really means. *Forrester Research*, May 2011.

[Far11] Sadri Fariba. Ambient intelligence: A survey. *ACM Comput. Surv.*, 43(4):36:1–36:66, October 2011.

[Fee09] Marc Feeley. The Gambit Scheme system, September 2009. http://dynamo.iro.umontreal.ca/~gambit.

[FFK10] Matthias Felleisen, Kathi Fisler, and Shriram Krishnamurthi. How to design worlds: Imaginative programming in DrScheme., June 2010. http://world.cs.brown.edu.

[FFP09] Matthew Flatt, Robert Bruce Findler, and PLT. Guide: PLT scheme. Introduction PLT-TR2009-guide-v4.2.3, PLT Scheme Inc., December 2009. http://plt-scheme.org/techreports/.

[FW08] Daniel P. Friedman and Mitchell Wand. *Essentials of Programming Languages, 3rd Edition*. The MIT Press, 3 edition, 2008.

[GCM+11] Sebastián González, Nicolás Cardozo, Kim Mens, Alfredo Cádiz, Jean-Christophe Libbrecht, and Julien Goffaux. Subjective-c: bringing context to mobile platform programming. In *Proceedings of the Third international conference on Software language engineering*, SLE'10, pages 246–265, Berlin, Heidelberg, 2011. Springer-Verlag.

[Geo09] Cant Geoff. A portable Objective-C bridge for Common Lisp, September 2009. http://www.common-lisp.org/project/cl-objc/.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[GL95] Wolfgang Golubski and Wolfram-M. Lippe. A complete semantics for Smalltalk-80. *Computer Languages*, 21(2):67–79, July 1995.

[GMH07] Sebastián González, Kim Mens, and Patrick Heymans. Highly dynamic behaviour adaptability through prototypes with subjective multimethods. In *Proceedings of the 2007 symposium on Dynamic languages*, DLS '07, pages 77–88, New York, NY, USA, 2007. ACM.

[GOBK79] Birtwhistle G.M., Dahl O.J., Myhrhaug B., and Nygaard K. *Simula Begin*. Chartwell-Bratt Ltd, 1979.

[Gon08] Sebastián González. *Programming in Ambience: Gearing Up for Dynamic Adaptation to Context*. PhD thesis, Université catholique de Louvain, 2008.

[Goo09] Google Inc. Google Maps for mobile, September 2009. http://www.google.com/mobile/products/maps.html.

[Gra93] Paul Graham. *On LISP: Advanced Techniques for Common LISP*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[Gro03] IST Advisory Group. Ambient intelligence: from vision to reality, September 2003.

[GWDD06] Kris Gybels, Roel Wuyts, Stéphane Ducasse, and Maja D'Hondt. Inter-language reflection: A conceptual model and its implementation. *Comput. Lang. Syst. Struct.*, 32(2-3):109–124, July 2006.

[Gyb03] Kris Gybels. SOUL and Smalltalk - just married: Evolution of the interaction between a logic and an object-oriented language towards symbiosis. In *Proceedings of the Workshop on Declarative Programming in the Context of Object-Oriented Languages*, 2003.

[Hal85] Robert H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, October 1985.

[HCH08] Robert Hirschfeld, Pascal Costanza, and Michael Haupt. Generative and transformational techniques in software engineering ii. chapter An Introduction to Context-Oriented Programming with ContextS, pages 396–407. Springer-Verlag, Berlin, Heidelberg, 2008.

[HCN08] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology, March-April 2008, ETH Zurich*, 7(3):125–151, 2008.

[HDM93] Robert Harper, Bruce F. Duba, and David B. Macqueen. Typing first-class continuations in ml. *Journal of Functional Programming*, 3:465–484, 1993.

[HFW86] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Obtaining coroutines with continuations. *Computer Languages, Systems Structures*, 11:143–153, 1986.

[HMPJH05] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 48–60, New York, NY, USA, 2005. ACM.

[Hoa83] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 26(1):53–56, January 1983.

[HPSA10] Robert Hirschfeld, Michael Perscheid, Christian Schubert, and Malte Appeltauer. Dynamic contract layers. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2169–2175, New York, NY, USA, 2010. ACM.

[Hum12] Todd Humphreys. The GPS Dot and its discontents: Privacy vs. GNSS integrity. *InsideGNSS*, 7(2):44–48, 2012.

[Joh96] Ousterhout John. Why threads are a bad idea (for most purposes). In *USENIX Winter Technical Conference*, January 1996.

[KAM11] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. Eventcj: a context-oriented programming language with declarative event-based context transition. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 253–264, New York, NY, USA, 2011. ACM.

[Ken96] Dybvig R. Kent. *The Scheme Programming Language: ANSI Scheme*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1996.

[Koc09] Stephen Kochan. *Programming in Objective-C 2.0*. Addison-Wesley Professional, 2009.

[KR91] Gregor Kiczales and Jim Des Rivieres. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

[Kri09] Shriram Krishnamurthi. The Moby Scheme compiler for smartphones or, is that a parenthesis in your pocket? In *Proceedings of the International Lisp Conference (ILC 2009)*, 2009.

[LASH11] 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.

[Lea99] Doug Lea. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[LML+10] Nicholas D. Lane, Emiliano Miluzzo, Hong Lu, Daniel Peebles, Tanzeem Choudhury, and Andrew T. Campbell. A survey of mobile phone sensing. *Comm. Mag.*, 48(9):140–150, September 2010.

[LMVD10] Andoni Lombide Carreton, Stijn Mostinckx, Tom Van Cutsem, and Wolfgang De Meuter. Loosely-coupled distributed reactive programming in mobile ad hoc networks. In *Objects, Models, Components, Patterns, 48th International Conference, TOOLS 2010, Málaga, Spain, June 28 - July 2, 2010. Proceedings*, pages 41–60, 2010.

[Lom11] Andoni Lombide Carreton. *Ambient-Oriented Dataflow Programming for Mobile RFID-Enabled Applications*. PhD thesis, Vrije Universiteit Brussel, 2011.

[LS88] Barbara Liskov and Liuba Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 260–267, New York, NY, USA, 1988. ACM.

[Mar80] Chris D. Marlin. Coroutines: A programming methodology, a language design and an implementation. *Lecture Notes in Computer Science*, 1980.

[MCC98] Ernst Michael, Kaplan Craig, and Chambers Craig. Predicate dispatching: A unified theory of dispatch. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, ECOOP '98, pages 186–211, London, UK, UK, 1998. Springer-Verlag.

[Mey92] Bertrand Meyer. Applying "design by contract". *Computer*, 25:40–51, October 1992.

[MF08] Jacob Matthews and Robert Bruce Findler. An operational semantics for Scheme. *J. Funct. Program.*, 18(1):47–86, January 2008.

[MGB+09] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: a programming language for ajax applications. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 1–20, New York, NY, USA, 2009. ACM.

[MI09] Ana Lúcia De Moura and Roberto Ierusalimschy. Revisiting coroutines. *ACM Transactions on Programming Languages and Systems*, 31:1–31, 2009.

[Mil04] Todd Millstein. Practical predicate dispatch. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '04, pages 345–364, New York, NY, USA, 2004. ACM.

[MR97] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Trans. Comput. Syst.*, 15(3):217–252, August 1997.

[MRI04] Ana Lúcia De Moura, Noemi Rodriguez, and Roberto Ierusalimschy. Coroutines in lua. *Journal of Universal Computer Science*, 10:925, 2004.

[MRO10] Ingo Maier, Tiark Rompf, and Martin Odersky. Deprecating the Observer Pattern. Technical report, 2010.

[NBF96] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads programming.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.

[PBV+10] Koosha Paridel, Engineer Bainomugisha, Yves Vanrompay, Yolande Berbers, and Wolfgang De Meuter. Middleware for the internet of things, design goals and challenges. *ECEASST*, 28, 2010.

[Ped03] Pinto Pedro. Dot-Scheme: A PLT Scheme FFI for the .NET framework. In *Proceedings of the Scheme and Functional Programming Workshop*, 2003.

[PyO09] PyObjC Project. PyObjC: The Python <-> Objective-C bridge, September 2009. http://pyobjc.sourceforge.net/.

[R.98] Pucella Riccardo R. Reactive programming in standard ML. In *Proceedings of the 1998 International Conference on Computer Languages*, ICCL '98, pages 48–, Washington, DC, USA, 1998. IEEE Computer Society.

[SA05] Lee Salzman and Jonathan Aldrich. Prototypes with multiple dispatch: an expressive and dynamic object model. In *Proceedings of the 19th European conference on Object-Oriented Programming*, ECOOP'05, pages 312–336, Berlin, Heidelberg, 2005. Springer-Verlag.

[SAW94] Bill Schilit, Norman Adams, and Roy Want. Context-aware computing applications. In *Proceedings of the 1994 First Workshop on Mobile Computing Systems and Applications*, pages 85–90, Washington, DC, USA, 1994. IEEE Computer Society.

[SBS04] Manuel Serrano, Frédéric Boussinot, and Bernard Serpette. Scheme fair threads. In *Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '04, pages 203–214, New York, NY, USA, 2004. ACM.

[Sch12] Simon De Schutter. Stress Testing Language Prototyping Methodologies: Defining a Dialect of Racket Featuring Interruptible Computations. Master's thesis, Vrije Universiteit Brussel, Brussels, Belgium, June 2012.

[SDF+09] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton Van straaten, Robby Findler, and Jacob Matthews. Revised6 report on the algorithmic language scheme. *J. Funct. Program.*, 19:1–301, August 2009.

[Sei04] Peter Seibel. *Practical Common Lisp*. Apress, 2004.

[SGP12a] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. Context-oriented programming: A software engineering perspective. *Journal of Systems and Software*, 85(8):1801 – 1817, 2012.

[SGP12b] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. Contexterlang: introducing context-oriented programming in the actor model. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, AOSD '12, pages 191–202, New York, NY, USA, 2012. ACM.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[Tan08] Éric Tanter. Contextual values. In *Proceedings of the 2008 symposium on Dynamic languages*, DLS '08, pages 3:1–3:10, New York, NY, USA, 2008. ACM.

[Tan09] Éric Tanter. Beyond static and dynamic scope. In *Proceedings of the 5th symposium on Dynamic languages*, DLS '09, pages 3–14, New York, NY, USA, 2009. ACM.

[TCW+12] 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 *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 418–425, New York, NY, USA, 2012. ACM.

[Val11] Jorge Vallejos. *Modularising Context Dependency and Group Behaviour in Ambient-oriented Programming*. PhD thesis, Vrije Universiteit Brussel, 2011.

[Van08] Tom Van Cutsem. *Ambient References: Object Designation in Mobile Ad Hoc Networks*. PhD thesis, Vrije Universiteit Brussel, 2008.

[VCMDM07] Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. Linguistic symbiosis between actors and threads. In *Proceedings of the 2007 international conference on Dynamic languages: in conjunction with the 15th International Smalltalk Joint Conference 2007*, ICDL '07, pages 222–248, New York, NY, USA, 2007. ACM.

[VGBB+08] Jorge Vallejos, Elisa Gonzalez Boix, Engineer Bainomugisha, Pascal Costanza, Wolfgang De Meuter, and Éric Tanter. Towards resilient partitioning of pervasive computing services. In

Proceedings of the 3rd ACM workshop on Software engineering for pervasive services, SEPS '08, pages 15–20, New York, NY, USA, 2008. ACM.

[VGC+10] Jorge Vallejos, Sebastián González, Pascal Costanza, Wolfgang De Meuter, Theo D'Hondt, and Kim Mens. Predicated generic functions: enabling context-dependent method dispatch. In *Proceedings of the 9th international conference on Software composition*, SC'10, pages 66–81, Berlin, Heidelberg, 2010. Springer-Verlag.

[vLDN07] Martin von Löwis, Marcus Denker, and Oscar Nierstrasz. Context-oriented programming: beyond layers. In *Proceedings of the 2007 international conference on Dynamic languages: in conjunction with the 15th International Smalltalk Joint Conference 2007*, ICDL '07, pages 143–156, New York, NY, USA, 2007. ACM.

[Wei93] Mark Weiser. Some computer science issues in ubiquitous computing. *Commun. ACM*, 36(7):75–84, July 1993.

[Wei95] Mark Weiser. Human-computer interaction. chapter The computer for the 21st century, pages 933–940. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995.

[WH00] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–252, 2000.