Interruptible Context-dependent Executions

A Fresh Look at Programming Context-aware Applications

Engineer Bainomugisha, Jorge Vallejos, Coen De Roover
Andoni Lombide Carreton and Wolfgang De Meuter

Software Languages Lab, Vrije Universiteit Brussel, Brussels, Belgium
{ebainomu, jvallejo, cderoove, alombide, wdmeuter}@vub.ac.be

Abstract

Context-aware applications provide end-users with enhanced experiences by continuously sensing their environment and adapting their behaviour to match the current context of use. However, developing *true* context-aware applications remains notoriously difficult due to the unpredictable nature of context changes. A context change may occur at any moment during a procedure execution, which may require an ongoing execution to be *promptly interrupted* in order to prevent the procedure from running in a wrong context. Currently, developers have to manually constrain a procedure execution to a particular context and take care of saving and restoring the execution state between context changes. Such manual approaches are error-prone and may lead to incorrect application behaviour.

This paper presents a novel programming language model called *interruptible context-dependent executions*, where a procedure execution is always constrained to happen only under a specified context. In this model, a procedure execution can be seamlessly interrupted or resumed depending on the context. Additionally, the procedure execution state is automatically preserved between interruptions. We present the *Flute* language that supports interruptible context-dependent executions.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Control structures

General Terms Design, Languages

Keywords Context-aware applications, interruptible context-dependent executions, reactive dispatching, prompt adaptations

Onward! 2012, October 19–26, 2012, Tucson, Arizona, USA.

1. Introduction

Context-aware applications continuously sense their environment in order to dynamically adapt their behaviour to match the current context of use (e.g., current location and user preferences). However, using the current programming languages, developing true context-aware applications remains notoriously difficult mainly because of the unpredictable nature of context changes.

More concretely, 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. An application should be able to automatically save its execution state between context changes and resume seamlessly from where it left off when it goes back to the previous context at a later point. 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). 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.

Over the past years researchers have carried out investigations on middleware [4, 17] and language approaches [11, 26, 31, 32] in order to ease the development of context-aware applications. For instance, a programming language approach called *context-oriented programming* (COP) [16], has recently been explored in a number of languages [7, 18] as a technique for expressing context-dependent behaviours in a program. COP provides linguistic abstractions that facilitate the dynamic adaptation of the application behaviour. However, in those approaches it is not possible to constrain

an entire procedure execution to a particular context. Once a procedure is selected and its execution is started, any context changes that occur during its execution cannot immediately affect the application behaviour. Such approaches are not suitable for cases where a procedure execution may need to be promptly adapted at any moment during the execution.

The **main contribution** of this paper is a novel programming language model called *interruptible context-dependent executions*, where a procedure execution is always constrained to happen only 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 a new programming language called *Flute* that adheres to the interruptible context-dependent executions model.

The Flute language allows 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 allows 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.

2. Motivating Scenarios

In this section, we introduce scenarios that motivate the need for a new programming language model for interruptible context-aware applications.

2.1 A Mobile Platform for Interruptible and Context-aware Applications

The increasing availability of context sources (such as GPS, proximity and accelerometer) on mobile devices has enabled developing applications that dynamically adapt their behaviour to match the current situation (e.g., location-specific services and user-specific services).

Consider, for instance, a mobile platform that provides a suite of applications used for different tasks. In order to alleviate the user from the burden of manually selecting which application to run for a task, such a mobile platform can be enhanced with context-awareness to automatically present to the user the application that is appropriate for the task at hand and the context of use. Thus, as the user moves about with their device, the mobile platform automatically switches back-and-forth between applications. Naturally, when there is a switch between applications, the user expects applications to automatically save their current state and be able to resume running from where they left off at a later point in time.

Applications running on such a platform can be enhanced with context-awareness to dynamically adapt their behaviour to match the changing user's needs. Below we present examples of such context-aware applications.

2.1.1 A Context-aware Calendar Application

Consider a context-aware calendar application that automatically launches to show the calendar items whenever the user moves within range of their 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. Therefore, when the owner is not the one using the device, it can dynamically adapt to show only the calendar items that are public. 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.

2.1.2 A Context-aware Printer Assistant

Consider a context-aware printer assistant application that provides functionalities for monitoring a printer's status (e.g., toner and paper levels) and managing printing tasks. Such an application can be specified to automatically launch on a mobile device whenever the user walks into a printer room or is nearby a printer. In addition, the printing tasks can be 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, the application can automatically pause the printing whenever another person walks into the printer room and resumes where it left off when that person leaves.

2.1.3 A Context-aware Task Guide

Our third motivating example is a mobile application for assisting individuals with memory impairments to

perform a sequence of tasks (e.g., making coffee, preparing soup, doing laundry or operating a TV). The application can be enhanced with context-awareness such that when the user is at home, the application is automatically launched and presents to the user a guide on how to perform a certain task depending on the user's needs. For instance, when the user is nearby a coffee machine, the application automatically offers to assist the user through the steps and instructions of making a coffee. Similarly, when the user walks into the kitchen the system presents steps on how to prepare soup. When the user leaves the kitchen or moves away from the coffee machine before the completion of a task sequence, the application is interrupted but automatically resumes from where it left off when the user moves back such that he/she can continue with the task. These steps and instructions may be further enhanced with context-awareness to vary depending the context such as the user's experience.

2.2 Scenarios Analysis

Context-aware applications such as the ones described above exhibit new characteristics that sets 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*.

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 if the specified context conditions are satisfied and should not be allowed to execute in a 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, when a user is running a task guide application on his/her mobile device, the execution of the sequence of tasks to operate a TV should only happen when the user is nearby a TV but not when the user is in the kitchen. Similarly, in the context-aware printer assistant application, the execution of the printing task for confidential documents should only proceed when there is no other person in the printer room.

Prompt adaptability. Context-aware applications need to promptly and continuously adapt their behaviour to match the current context. A single context-aware application is typically composed of variants of behaviours that need to be dynamically made available depending on the context. As such, the application needs to ensure that when there is a context change the correct variant of the application is promptly made available without explicit user intervention. For instance, when the user is navigating through a calendar application on his/her mobile device and gives his/her device to a coworker, the calendar application should promptly adapt the current display and provide the coworker with a calendar display that includes only public calendars and any other public display preferences such as the background colour.

Sudden interruptibility. Context-aware applications need to always be prepared for sudden interruptions 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, interruptions are the norm in context-aware applications. If there is a context change while an application is running, another application that match the current context may start running while interrupting the previous application. For instance, in the above scenarios, when the user walks into a printer room while running a calendar application on his/her mobile device, the calendar application needs to be temporarily interrupted and the printer assistant application should be started. Similarly, when the user leaves the printer room, and say, walks back to his/her office, the printer application should be interrupted and the calendar application should automatically resume from where it left off.

We would like to stress that the these characteristics are not specific to the above scenarios but depict a general pattern that is present in most context-aware applications. However, the lack of suitable programming language abstractions coupled with the unpredictable nature of context changes renders the task of developing *true* context-aware applications notoriously difficult. Because context changes can occur at any moment during the execution of a procedure, it is possible that a procedure execution that started in a correct context may end up running in a 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). To prevent that, the developer must perform 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 can lead to negative effects on program comprehension and maintainability (e.g., introducing a new context source implies modifying all the existing context checks). In addition, the developer must 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 must devise means to capture and restore the procedure execution

state. Such concerns are not trivial and it is almost impossible to express them manually.

To concretely illustrate these issues, we consider a language like Scheme [30] to express the behaviour for the steps of making a coffee in the context-aware task guide application. In Scheme, we can define such a procedure as follows:

```
1 ;a task guide for making a coffee
2 (define making-coffee
3     (lambda (person-name)
4       (write "Welcome: " person-name)
5       (task "1.place a cup")
6       (task "2.select ingredients")
7       (task "3.press make button")
8       (task "4.pick your coffee")))
```

The above code snippet shows the definition of the `making-coffee` procedure that presents to the user a sequence of tasks for making a coffee. The above `making-coffee` procedure is context-**un**aware, i.e., it does not take into account any context changes and assumes that its execution runs uninterrupted from the start to the end. Thus, assume that this procedure is started on a mobile device when the user is nearby a coffee machine, the procedure will continue its execution even if the user moves away from the coffee machine, which is undesired.

In order to make the execution of the `making-coffee` procedure *context-aware*, the developer must manually insert checks within the body of the procedure, to ensure that its execution happens only in a particular context (i.e., only when the user is nearby a coffee machine). For instance, the execution of the `making-coffee` procedure may be naively enriched with context-awareness by redefining it as follows:

```
1  (define making-coffee
2      (lambda (person-name)
3        (if (nearby-coffee-machine?)
4            (write "Welcome: " person-name)
5            (save/suspend))
6        (if (nearby-coffee-machine?)
7            (task "1.place a cup")
8            (save/suspend))
9        (if (nearby-coffee-machine?)
10           (task "2.select ingredients")
11           (save/suspend))
12       (if (nearby-coffee-machine?)
13           (task "3.press make button")
14           (save/suspend))
15       (if (nearby-coffee-machine?)
16           (task "4.pick your coffee")
17           (save/suspend))))
```

This code example shows a redefined version of the `making-coffee` procedure with condition checks to constrain its execution to the correct context (i.e., it should be executed only when the user is nearby a coffee machine). To simplify the implementation, we assume that there is a language construct `save/suspend` that enables saving a procedure execution state and suspending the ongoing execution[1]. The `write` procedure displays a message to the user's mobile device while the `task` procedure displays the details of the task to be performed for making a coffee.

In order to ensure that the `making-coffee` procedure executes only if the user is nearby the coffee machine, the `(nearby-coffee-machine?)` condition is inserted before every expression in the procedure body (Lines 3, 6, 9, 12, and 15). If the context condition is *false* (i.e., if the user is not nearby the coffee machine), then the procedure execution is saved and suspended using the `save/suspend` construct (Lines 5, 8, 11, 14, and 17).

The context-aware version of the `making-coffee` procedure is visibly convoluted because of the verbose code that is needed to check the context conditions, save the execution state and suspend the execution. Clearly, implementing context-aware applications in this style is hard and error-prone. Even with all the checks and manual execution state management in the context-aware version of the `making-coffee` procedure, the above implementation is lacking since it does not include the logic of resuming the execution when the user moves back near to the coffee machine. These observations have motivated our vision for a new programming language model, *interruptible context-dependent executions*, which we present in the next section.

3. Interruptible Context-dependent Executions

In this section, we present the main ingredients of the *interruptible context-dependent executions* model. These ingredients are motivated by the scenarios that we described in Section 2.

3.1 Terminology

Before unveiling the model, we will first define the key terms that are used throughout this paper.

- The term *execution* is used to refer to a running procedure.
- The term *context-dependent execution* is used to refer to an execution that is constrained to run under particular context conditions.
- The term *execution state* is used to refer to the program counter (i.e., the rest of the expressions to be evaluated) and local bindings of an execution.

The remainder of this section presents the main ingredients of the interruptible context-dependent executions model. These are: *predicated procedures, reactive*

[1] The `save/suspend` construct can be easily implemented in languages that provide support for first-class continuations. For instance, in Scheme such a construct can be built on top of the native `call-with-current-continuation` function.

dispatching, interruptible and resumable executions, and scoped state changes.

3.2 Predicated Procedures

A context-aware application consists of procedures that define behavioural variations for different contexts. Each context-dependent procedure should be associated with a context predicate that specifies when the procedure is allowed to execute. The context predicate should be *implicitly* checked throughout the procedure execution in order to ensure that the execution happens only in the correct context. Traditional ways of associating a predicate to procedure using conditional statements are impractical because they require the developer to *explicitly* insert several `if` statements in the procedure body. That solution would be too cumbersome and could result in writing programs in a style where every statement in the procedure body is preceded with a context predicate. Therefore, a programming language that supports the interruptible context-dependent executions model, should provide the developer with a construct to associate a context predicate a procedure and the language runtime should ensure that the context predicate is satisfied throughout the execution.

3.3 Reactive Dispatching

The execution of a context-aware application involves a dispatching process to determine the appropriate context-dependent 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 context changes continuously occur, implies that the applicability of a context-dependent procedure to execute depends on a context predicate that may change dynamically. This implies that a context-dependent 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 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 [9] where the selection of the applicable procedure happens once and is based only on the currently available information.

3.4 Interruptible Executions

The execution of a context-dependent procedure should be constrained to happen only under a particular context condition. This requires that a context-dependent procedure starts or continues executing only if the specified context predicate is satisfied. If the context predicate is no longer satisfied while its associated procedure execution is ongoing, then the execution should be interrupted. A programming language for interruptible context-dependent executions should provide the developer with interruption strategies to specify what to do (depending on the application). We identify two interruption strategies.

The execution is suspended. When the context predicate is no longer satisfied, the corresponding execution is paused and its execution state is saved. Pausing an execution means that it is possible to resume the execution later on if its associated context predicate 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.

The execution is aborted. Another interruption strategy is to abort the execution once 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 shared or global variables before the execution is aborted may need to be undone. We further explore the management of state changes that arise in interruptible executions in Section 3.6.

3.5 Resumable Executions

Another important consideration that needs to be taken into account is what to do with a previously interrupted execution whose associated context predicate later becomes satisfied again. As context changes continually occur, 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. Therefore, a programming language for interruptible context-dependent executions should provide the developer with resumption strategies to re-establish a previously interrupted execution. We can identify two resumption strategies (the choice depends on the application).

The interrupted execution is resumed. A previously suspended execution can be resumed such that it continues from the exact point where it left off before interruption. The execution state should be restored to the same program instruction. Once the execution is resumed, the context predicate should be checked again

throughout the execution. This ensures that executions can be seamlessly suspended and resumed depending on the current context of use.

The interrupted execution is restarted. Another strategy is to restart the suspended execution 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*.

The resumption process should be event-driven (i.e., triggered by the availability of new and relevant context changes) in order to avoid unnecessary re-evaluations of context predicates even when the relevant context sources have not received new values.

3.6 Scoped State Changes

The execution of a context-dependent procedure may result in state changes[2] to the values stored in shared or global variables. The fact that a context-dependent execution can be suspended or resumed at a later moment, may result in situations where state changes made by one execution become visible to other executions. This can lead to undesirable behaviour (e.g., observing inconsistent values between suspension time and resumption time). It is therefore necessary that a programming language for the context-dependent executions model, provides mechanisms to enable the developer to *scope the visibility of state changes*. We identify three state management strategies that the developer can select from to scope state changes among executions.

Immediate visibility. With this strategy, changes made by one execution to a shared state are immediately visible by other executions that share this state.

Deferred visibility. This strategy ensures that state changes remain local to the execution and become visible to other executions on completion of the execution. This concept is comparable to software transactions [14, 28] and side effects management techniques of the *worlds* construct [34].

Isolated state changes. This strategy guarantees isolation of state changes. That is, any state changes made by one execution are restricted to that execution and are not visible by other executions.

4. The Flute Language

We now present a programming language called *Flute* that adheres to the interruptible context-dependent executions model proposed in Section 3. The Flute language has been implemented as a meta-interpreter in iScheme [2], our Scheme implementation that runs on iOS devices.

[2] 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.

4.1 Building Blocks: Modes and Modals

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

Definition 1. ***(Mode)*** *A mode defines a variant of behaviour (context-dependent procedure) or state (context-dependent variable) for a particular context. It is associated with a context predicate to specify the context conditions in which it is constrained to run.*

Definition 2. ***(Modal)*** *A modal is a group of related modes. It specifies context sources that may affect the execution of those modes.*

In Flute, context-dependent procedures and context-dependent variables are represented as a suite of modes, each defining a different behaviour or value for a different context. Related context-dependent procedure or variable 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. Flute 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. 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 interruptible context-dependent executions by means of illustrative examples. We will use the context-aware calendar application as the running example throughout this section. Figure 1 shows the informal description of the Flute syntax for mode, modal, and context source definitions.

4.2 Modes of a Variable

In Flute, a variable has one or more values (modes) that correspond to different 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, while developing a 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, such a context-dependent variable can be expressed as follows.

```
;A general form for a context source definition
(define <context-source-name>  (ctx-event))

;A general form for a modal definition
(define <modal-name> (modal (<context-sources>)))

;A general form for a variable mode definition
(mode (<modal-name>)
    <context-predicate>
    <value-expression>)

;A general form for a procedure mode definition
(mode (<modal-name>)
   <context-predicate>
   (<configuration-options>)
   (lambda (<parameters>)
       <body>))
```

Figure 1. An informal description of the Flute syntax for modal, mode, and context source definitions.

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

Listing 1. Defining variable modes

Listing 1 creates `bg-colour` as a modal variable that has two modes. A modal is created using the special form `modal` while a mode is created using the special form `mode`. In addition, the modal definition specifies a context source upon which context predicates operate. A context source is created using the special form `ctx-source`. 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 sensors (such as GPS) that are available on a mobile device are discussed in Section 4.4. Each mode definition specifies the modal it belongs to, a context predicate and a value for the mode when the context predicate is *true*. In the remainder of this section we explain the variable access and assignment semantics of modal variables in Flute.

4.2.1 Variable Access Semantics

A modal variable can be accessed like a regular variable in a programming language by using the variable name. The difference, however, is that accessing a modal variable can yield a different value depending on the current context of use. For instance, in the above example, accessing the variable `bg-colour` may yield the colour value as `brown-colour` or `grey-colour` depending on the current device user. Below we illustrate the semantics of accessing the `bg-colour` in different contexts. The input expression is prefixed with a `>` 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
```

As the above example shows, accessing the `bg-colour` when the current user is the device owner, yields a `grey-colour` value, and yields `brown-colour` when the user is not the device owner. In Section 4.3, we will see that abstraction for variable modes is useful for developing context-aware applications because often there is a need for a data structure that holds different values for different contexts. Observe that it is possible to add new modes of a variable at runtime, and when they are added, they become part of the suite value modes for the modal variable. This has an advantage that developers can add unanticipated variable modes on demand. Note that in case there are multiple values (i.e., if there are more than one context predicates that are satisfied) or there is no value found (i.e., if there is no context predicate that is satisfied), an exception is thrown. We further discuss other dispatching semantics of procedure modes in Section 4.3.1.

4.2.2 Assignment Semantics

Performing an assignment on a modal variable only affects the value of the variable mode whose context predicate evaluates to *true*. Like with the variable access, before performing a state change to the value, the correct mode is looked up depending on the current context. Below we illustrate an example of mutating the `bg-colour`.

```
1  ;suppose the current user is the device owner
2  >  (set! bg-colour blue-colour)
3
4  ;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 snippet, performing an assignment operation on the `bg-colour` variable

when the current device user is the owner, only affects the value of that mode.

4.3 Modes of a Procedure

Context-dependent procedures in Flute are expressed in terms of different modes of a modal. For instance, the context-aware calendar application consists of two 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. In Flute, we can express such variations of modes as follows.

```
1  ;modal definition
2  (define agenda (modal (current-user)))
3
4  ;shared variables
5  (define date-range          2)
6  (define display-scale       4)
7
8  ;configuration options definition
9  (define config
10   (create-config suspend resume isolated))
11
12 ;mode definition
13 (define show-private-agenda
14   (mode (agenda)
15     (owner? current-user) ;context predicate
16     (config)              ;configuration options
17     (lambda ()
18       (write "private agendas")
19       (set! display-scale 8)
20       (write bg-colour)
21       (scale display-scale)
22       ...
23       (write calendars))))
24
25 ;mode definition
26 (define show-public-agenda
27  (mode (agenda)
28    (not-owner? current-user)
29    (default-config)
30    (lambda ()
31      (write "public agendas")
32      (write bg-colour)
33      ...
34      (write calendars))))
35
36 (agenda)
```

Listing 2. Defining procedure modes

Listing 2 creates a modal `agenda` using the special form `modal` and two modes `show-private-agenda` and `show-public-agenda` using the special form `mode`. As in the case of modal variables, the `current-user` variable in the modal definition, specifies the context source. `date-range` (on Line 5) and `display-scale` (on Line 6) are shared variables that are visible to both modes. The `date-range` value specifies the date range of calendar items to display. The `display-scale` value specifies the scale of the font size of the calendar display. As with modes of a variable, new procedure modes can be dynamically added to a modal on demand. Each mode definition includes a context predicate that must be satisfied throughout the execution of the mode. In the above example, the context predicates for the private and public modes are `(owner? current-user)` and `(not-owner? current-user)`, respectively. To avoid ambiguities, the developer should ensure that context predicates are mutually exclusive.

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 9 while the public agenda mode uses the default configuration. For conciseness, the above implementation does not include the graphical user interface (GUI) concerns of the calendar application. The screenshot of the context-aware calendar application running in the public mode on an iPad device can be found in the Appendix (Figure 3).

A context predicate specified in each mode plays two roles. First, it is used by the dispatcher to select the applicable mode (dispatching) to execute for the current context of use. Second, it is also used by the runtime to ensure that the mode execution continues to happen in the correct context by continually evaluating the context predicate at every evaluation step of the procedure of body for the mode.

Let us first explain the dispatching process.

4.3.1 Reactive Dispatching of Modes

The execution of modes is initiated by invoking a modal. Invoking a modal requires a dispatching mechanism to select the applicable mode to execute for the current context. For instance, in Listing 2 invoking the agenda modal as `(agenda)`, may execute a private agenda mode procedure or the public agenda mode procedure depending on the current context.

The dispatcher starts by evaluating all context predicates that are associated with the modes that belong to the same 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 typically occur continuously, it is possible that some context predicates that could not be satisfied may become satisfied later and thus requiring their associated modes to be executed. In Flute, the dispatcher is implicitly registered to the context sources that may affect the context predicates and the dispatching process is triggered again 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 procedure-not-found exception* because *the mode may later be found* when relevant context changes are observed.

Note that binding a procedure mode to a variable is optional. However, if a mode is bound to a variable, it is possible to invoke the 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. Allowing modes to be directly invoked also makes it possible to define recursive modes. In the next section, we discuss the execution semantics of procedure modes.

4.3.2 Interruptible Execution of Modes

Once the dispatcher selects the mode to execute, its associated context predicate must be satisfied throughout the mode execution, otherwise the execution is interrupted based on the specified interruption strategy. For instance, in the calendar example, suppose that the agenda modal is initially invoked when the current user is the device owner, thus the `(owner? current-user)` predicate will be satisfied. As a result, the execution of the private agenda mode will be started and the user will view all the agenda items including private appointments and office meetings. Suppose that the user then gives the device to another user while the private mode is executing. As a consequence, there will be a context switch and `(owner? current-user)` will become *false* and therefore, the execution of the private agenda mode will be promptly *suspended* since its configuration options `config` specify `suspend` as the interruption strategy. On the other hand, the context predicate `(not-owner? current-user)` will be satisfied and therefore, the public mode execution will be started.

4.3.3 Event-driven Resumption of Suspended Executions

As context changes occur, context sources will receive new values and as a result any suspended executions whose context predicates operate on those context sources will be scheduled for resumption. For instance, in the calendar example, suppose that while the public agenda mode is executing, the device is given back to the owner. Then the execution of the public agenda mode will be promptly interrupted. Conversely, the execution of the 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. Subsequently, the `(owner? current-user)` context predicate is re-evaluated and the previously suspended private agenda mode execution will be resumed.

4.3.4 Scoping State Changes

The configuration options include a strategy for controlling the visibility of the state changes made by a mode to the shared state. In Listing 2, the private agenda mode is specified with the `isolated` strategy, which implies that all state changes remain local to the mode. For instance, the private agenda mode modifies the shared variable `display-scale` variable to increase display scale of the agenda items (i.e., `(set! display-scale 8)` on Line 19). Such a state change will remain local to the private agenda mode and will not be visible to the public agenda mode. With the `isolated` strategy, the Flute runtime keeps a *local copy* when a shared variable is accessed for the first time, and any subsequent changes are made to the local copy. Other state changes scoping strategies are `immediate` and `deferred` as discussed in Section 3.6. 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.

4.3.5 Lexical and Dynamic Extent of Context Predicates

When a mode is internally defined within another mode, the context predicate of the enclosing mode must be respected throughout the execution of the internal mode. In addition, a context predicate of the enclosing mode must be respected throughout the execution of modes that are invoked from the mode body. We classify these as lexical and dynamic extent of context predicates.

Lexical extent of a context predicate. In Flute, a mode that is defined inside another mode "inherits" the context predicate of its enclosing mode as part of the context predicate under which the mode can exe-

cute. Consider for example, the context-aware calendar application that is specified with a context predicate to run only when the user walks into his/her office. This can be defined as a mode that encloses the private and public agenda modes as follows.

```
1  (define flute-apps (modal (location)))
2
3  (define calendar-assistant
4    (mode (flute-apps)
5      (office? location)
6      (default-config)
7      (lambda ()
8      ...
9
10   (define show-private-agenda
11     (mode (agenda)
12       (owner? current-user) ;a context predicate
13       (config)              ;configuration
             options
14       (lambda ()
15         ...)))
16
17   (define show-public-agenda
18    (mode (agenda)
19      (not-owner? current-user)
20      (default-config)
21      (lambda ()
22        ...))))))
```

Listing 3. Nested mode definitions.

Listing 3 shows the definition of the `show-private-agenda` and `show-public-agenda` modes as internal definitions of the `calendar-assistant` mode. The `calendar-assistant` represents the entire calendar application and is specified with the context predicate `(office? location)` 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 Flute platform. In this example, the `(office? location)` context predicate must also be satisfied during the execution of the the `show-private-agenda` and `show-public-agenda` modes (in addition to their own context predicates).

Dynamic extent of a context predicate. When a modal is invoked inside a mode, then the context predicate of the mode at the *callee* side must also be ensured throughout the execution of any subsequent direct and indirect invocations of modes.

4.3.6 Demarcating Uninterruptible Regions

The default semantics of Flute is that the execution of a mode may be interrupted at any evaluation step of its body expressions. However, it may be required that some critical sections of a procedure must run uninterrupted. For this, Flute provides a dedicated construct `(continuous` *<expressions>*`)` to demarcate such uninterruptible regions.

4.4 Defining Context Sources

Context sources in Flute are represented as *reactive values*. 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. For instance, the context source for the current location is defined as follows.

```
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 4. Defining context sources

Listing 4 shows the definition of the context source `gps-coordinates` which is created using the `ctx-event` construct. The `location` context source is created by applying the `gps->location` procedure on the `gps-coordinates` context source (Line 5). The `gps->location` procedure transforms raw GPS coordinates into a high-level context value such as office or home. The GPS coordinates are obtained using the `CURRENT-LOCATION` construct that is provided by iScheme. 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. This in turn automatically updates the `location` context source with the new value. Note that once `gps-coordinates` gets new values, the `gps->location` procedure is automatically invoked.

A reactive value employs a push-driven model, which means that any procedure that operates on its value is immediately notified as soon as a new value is received. A similar evaluation model is used in functional reactive programming (FRP) approaches [6]. This facilitates the implementation event-driven resumption of suspended executions and reactive dispatching. A suspended execution is encapsulated as a resumption handler that establishes a link between relevant context sources and the suspended execution. When the context source receives a new value, the resumption handler is automatically executed which in turn triggers the evaluation continuation of the procedure body. A similar technique is used to link the dispatcher to context sources.

5. Flute Implementation

The Flute language is implemented as a meta-interpreter in Scheme. We implement the interpreter in a flavour of Scheme called *iScheme* [2], our Scheme im-

plementation that runs on iOS devices (such as iPhone and iPad). iScheme supports a *linguistic symbiosis*[3] between Scheme and Objective-C language which makes it possible to access context source APIs provided by the iOS platform. The Flute interpreter is implemented in a continuation-passing style [10], and therefore, explicitly passes a *continuation* parameter along with the environment. 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.

Availability. The current implementation of the Flute interpreter is available at
`http://soft.vub.ac.be/~ebainomu/Flute/`

6. Motivating Scenarios Revisited

Let us go back to the scenarios introduced in Section 2 in order to illustrate how Flute can be used to implement such applications.

6.1 The iFlute Platform

As a case study, we have implemented a prototype application platform called *iFlute Platform* where interruptible context-aware applications can be deployed. The screenshot of the platform running on an iOS device can be found in the Appendix (Figure 2).

Context-aware applications (developed in the Flute language) can be deployed on the iFlute platform and are automatically launched depending on the current context of use. As an experiment, we have so far developed and deployed example applications, namely, a context-aware calendar application and a context-aware printer assistant, and a context-aware task guide.

6.2 Example: A Context-aware Task Guide

The context-aware task guide application can be implemented in Flute as follows.

```
(define task-manager
  (mode (flute-apps)
    (at-home? location)
    (default-config)
    (lambda ()

      ;a modal definition for task guide
      (define task-guide (modal (nearby-object)))

      ;a mode for making coffee
      (define making-coffee
        (mode (task-guide)
          (coffee-machine? nearby-object)
          (default-config)
          (lambda (person-name)
            (write "Welcome: " person-name)
            (task "1.place a cup")
            (task "2.select ingredients")
            (task "3.press make button")
            (task "4.pick your coffee"))))

      ;a mode for making soup
      (define making-soup
        (mode (task-guide)
          (soup-maker? nearby-object)
          (default-config)
          (lambda (person-name)
            (write "Welcome: " person-name)
            (task "1.select soup can")
            (task "2.get a pan")
            (task "3.pour soup in the pan")
            (task "4.turn on hot plate")
            ;demarcating a non-interruptible
                region
            (continuous
              (task "5.remove pan")
              (task "6.turn off hot plate"))
            (task "7.serve soup"))))

      ;a mode for operating a TV
      (define tv-controller
        (mode (task-guide)
          (tv? nearby-object)
          (default-config)
          (lambda (person-name)
            (write "Welcome: " person-name)
            (task "1.get remote controll")
            (task "2.enable TV mode")
            (task "3.turn on TV")
            (task "4.enable decoder mode")
            (task "5.turn on decoder")
            (task "6.select channel"))))

      (task-guide name))))
```

Listing 5. Implementing the context-aware task guide application in Flute

Listing 5 shows the implementation of the context-aware task guide application. `task-manager` is a mode that represents the entire task guide application and belongs to the `flute-apps` modal. The `task-manager` is specified with the context predicate `(at-home? location)`, which implies that the application should only run when the user is at home. The `task-manager` mode includes `making-coffee`, `making-soup` and `tv-controller` which belong to the `task-guide` modal. Each of those modes defines the behaviour of guiding a user through the steps of performing a certain task. In addition, each mode is associated with a context predicate to specify under what context condition the mode can be executed. For instance, the `making-coffee` mode is specified with `(cofee-machine? nearby-object)` predicate which means that the mode should be executed when the user is nearby a coffee machine. Since those modes are de-

[3] The language symbiosis between Scheme and Objective-C enables access to the Objective-C APIs from Scheme programs and vice versa.

fined inside the `task-manager` mode, the context predicate `(at-home? location)` must also be satisfied in order for the mode to be executed. Suppose that the user arrives at home, then the `task-manager` application will be executed. As the user moves about, if the user is nearby a coffee machine, then the task guide for making coffee will be presented to him/her. If the user moves about before the completion of the steps of the making coffee, the execution of the mode will be suspended and can later resume from the same step when the user moves back in range with the coffee machine. Tasks 5 and 6 of making soup are enclosed in the `continuous` block meaning that the execution cannot be interrupted at the step of the evaluation. In this example, this is necessary since removing a pan and turning off the hot plate tasks must be done immediately after each other to avoid leaving the hotplate on after removing the pan.

6.3 Example: A Context-aware Printer Assistant

```
1  (define printing-assistant
2    (mode (flute-apps)
3      (printer-room? location)
4      (default-config)
5      (lambda ()
6        ;a modal variable for documents
7        (define documents (modal (motion-detector)))
8
9        ;a modal procedure for printing modes
10       (define printing (modal (motion-detector)))
11
12       ;shared variable for paper level
13       (define paper-level (tray-load))
14       ...
15       ;variable modes for documents to print
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       ;procedure mode for confidential printing
25       (define confidentialprinting
26         (mode (printing)
27           (alone? motion-detector)
28           (default-config)
29           (lambda ()
30             (define print-queue documents)
31             (define header "Confidential")
32             (define owner  user-name)
33             ;loop over documents
34             ;add metadata and print
35             (for-each
36               (lambda (doc)
37                 (metadata header owner doc)
38                 (print doc))
39               print-queue))))
40
41       ;procedure mode for regular printing
42       (define regularprinting
43         (lambda (printing)
44           (people-nearby? motion-detector)
45           (default-config)
46           (lambda ()
47             (define print-queue documents)
48             ;loop over documents and print
49             (for-each
50               (lambda (doc)
51                 (print doc))
52               print-queue))))
53     ...
54     (render-toner-paper-status)
55     (printing))))
```

Listing 6. Implementing the context-aware printer assistant application in Flute.

Listing 6 shows the implementation of the context-aware printer assistant application in Flute. The `printing-assistant` mode belongs to the `flute-apps` modal and is specified with a context predicate `(printer-room? location)` which implies that the application will be launched when the user walks into a printer room. Line 7 creates the `documents` modal variable whose value is a list of either confidential or regular documents depending on the context when it is accessed (i.e., confidential documents the user is alone in the printer room and regular documents when 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 10 creates the `printing` procedure modal which consists of the `confidentialprinting` and `regularprinting` modes. Each mode is associated with a context predicate to specify when it should be executed. For instance, the `confidentialprinting` mode is associated with the `(alone? motion-detector)`, which implies that the mode should be executed only the user is alone in the printer room. Therefore, suppose the `confidential` mode is executing and another person walks into the printer room, the printing will be suspended and resumed when the person walks out of range. The screenshot of the context-aware printer assistant application running on the iPad device can be found in Appendix A Figure 4.

6.4 Discussion

In this section, we have demonstrated the Flute language in action by implementing the example context-aware applications. Flute enables the developer to constrain a context-dependent procedure execution to happen only under a particular context by means of a single context predicate. The context predicate is implicitly evaluated at every step of the execution of the procedure body. This ensures that the context predicate is respected throughout the procedure execution. If the context predicate is no longer satisfied, the procedure execution is *promptly interrupted* using the developer specified interruption strategy. Flute employs an event-

driven resumption mechanism that *promptly resumes* any previously interrupted executions when relevant context changes occur. In addition, Flute provides state scoping strategies that enable the developer to control the visibility of state changes among executions.

Such context-driven executions that promptly react to context changes cannot be achieved using the existing approaches. For instance, in the current context-oriented programming languages [7, 11, 18], once a procedure is selected and its execution is started, it is not possible to interrupt the execution. As a consequence the developer must manually guard each expression in the procedure body. Moreover, the developer has to explicitly capture and restore the procedure execution in order to preserve the execution state between context changes. Addressing such concerns manually is error-prone and can lead to incorrect application behaviour. We further discuss current approaches in Section 8.

7. Challenges

The interruptible context-dependent executions model and its first instantiation, the Flute language, is only a first step towards our vision for *a fresh look at programming language support for context-aware applications*. However, to make this vision a reality, there is still more work to do. We discuss some of the remaining challenges below.

Ambiguous context predicates. An open issue which is not specific to the Flute implementation but also common to predicate-based dispatching approaches is the *predicate ambiguity problem* [21]. The current implementation of 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 may be true at the same time. When the Flute dispatcher encounters such cases during the dispatching process, 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. For instance, the developer can specify ordering priorities that can used select one mode in case there are multiple context predicates that evaluate to *true*. Another design choice is to employ a dedicated ambiguity resolution mechanism [1].

Garbage collection of suspended executions. Allowing executions to be suspended raises a garbage collection challenge. How long should suspended executions be stored before being subjected to garbage collection? In the best case scenario every suspended execution will be resumed or restarted at a later point when its associated context predicate becomes satisfied again. However, in the worst case scenario, the context predicate associated with a suspended execution *may never* become satisfied again. Such executions should be automatically garbage collected by the language runtime. The main difficulty is due to the fact that it is not possible to a *priori* identify context predicates that will never be satisfied again. There are possible solutions that we have explored (but not presented in this paper) such as allowing the developer to specify a condition or a predefined timeout such that when it becomes satisfied and the corresponding execution that is still suspended is automatically garbage collected.

Explicit *versus* implicit shared variables. The state scoping strategies that we have discussed, are applied to all variables that are shared by the executions. However, this involves a possible overhead of keeping track of state changes for all the variables that belong to the shared scope. Moreover, the fact that new modes can be dynamically added to a modal, it renders it difficult to control the variables that can be accessed by the modes. To alleviate such a concern, we are exploring the possibility of allowing the developer to explicitly specify the variables that can be shared by the modes that belong to the same modal. For instance, the agenda modal can explicitly specify the shared variables for its modes as follows:

```
1  ;modal definition
2  (define agenda (modal (current-user)
3    ;shared variables
4    (define date-range          2)
5    (define display-scale       4)))
```

Listing 7. Explicit shared variables in a modal definition

8. Related Work

To the best of our knowledge, there is no existing approach that supports the ingredients of the interruptible context-dependent executions model. However, our model absorbs some of its ingredients from existing approaches. In particular, our work can be compared to existing works on context-oriented programming languages, continuations, coroutines, threads, and functional reactive languages. The remainder of this section gives a review of those approaches.

8.1 Context-oriented Programming

The context-oriented programming (COP) [16] paradigm has been recently proposed for the development of context-aware applications. Subsequently, a number of COP languages have been developed. These

include ContextL [7], Ambience [11] and EventCJ [18]. COP focuses on modularising context-dependent behaviours (procedure definitions) into entities (called *layers*), which can be dynamically activated or deactivated at runtime. Most COP languages provide explicit language constructs to activate or deactivate layers. For example, ContextL provides `with-layers` and `without-layer` as constructs to activate and deactivate layers respectively. Activating a layer makes certain procedures available while deactivating a layer makes certain procedures unavailable. However, in COP once a procedure execution is started it is not possible to react to any context changes that occur before the completion of the procedure execution. Thus, none of the COP approaches support interruption of an ongoing procedure execution. Contextual values [31] support variables similar to our modes of variables that hold different values depending on the context. However, they do not support modes of procedures and there is no notion of interruptible and resumable executions.

Most recently, Vallejos *et al.* [32] proposed *predicated generic functions*, which is a *predicate dispatching mechanism* that enables one to associate context predicates to context-dependent functions. Like in our approach, a function definition is associated with a context predicate expression. However, in their case the context predicate is only used to determine the most-specific function that should be invoked but once the selected function and its execution started, it cannot be interrupted. The context predicate is only checked at the function dispatch stage whereas in our approach the context predicate is checked at every evaluation step during the function execution. Moreover, the dispatching process happens *only once* i.e., it does not take into account of future context changes.

8.2 Continuations and Coroutines

Programming languages like Scheme [30] and Standard ML [13] provide support for *first-class continuations* [15], which enable the developer to capture the current execution state during procedure execution and return to that particular point later on. For instance, Scheme provides the `call-with-current-continuation` (commonly abbreviated as `call/cc`) construct. The `call/cc` construct reifies the current continuation into a first-class function that can be arbitrarily invoked later. In order to capture execution state using `call/cc`, the developer needs to explicitly identify which points in the procedure body that an execution state needs to be captured and when it can be interrupted or resumed. Expressing context-dependent interruptible executions in this style, implies that the developer must precede each statement in the procedure body with a conditional statement in order to check for the validity of certain context condition. Since a context change can potentially occur at any evaluation step during the execution it is difficult to determine the interruption points. In addition, the developer must maintain an inventory of continuations mapping to different context events. Our approach builds on the idea of first-class continuations to preserve execution state, but with the focus on providing transparent and automatic execution state management (i.e., without requiring the developer to explicitly use continuation constructs in order to capture or restore the execution).

Coroutines [5], which are available in languages such as Lua [23] and Simula [3] support a control transfer mechanism that allows a coroutine to pass control back and forth between coroutines. As summarised in [19] and later in [22], the main characteristics of a coroutine are: (i) the state local to a coroutine persists between successive calls, and (ii) 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. Therefore, when the current executing coroutine transfers control to another coroutine, its execution becomes suspended and the execution of the target coroutine is resumed. However, in coroutines approaches the developer must explicitly transfer control (typically expressed using the `resume` or `yield` construct) at certain points in the procedure body. However, if the transfer of control depends on runtime context conditions (which is the case in context-aware applications) it is not always possible to determine those points at development time. And thus the developer has to insert multiple control transfer constructs and guard each one of those with a context predicate. Also, coroutines only support suspend and resume semantics and the developer has no other interruption mechanisms such as abort or restart. Moreover, coroutines neither provide support for automatic resumptions nor mechanisms to scope state changes.

8.3 Threads

Traditional threads [12, 24] 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* [27]. However, the two categories of threads appear at the two extremes. On the one extreme, there is preemptive threads, which are non-deterministic and their suspension or resumption is entirely based on a preemptive scheduler. Preemptive threads are unsuitable for context-aware applications since the interruption of a context-dependent execution depends on certain context conditions and not on a predefined time-slot by the underlying scheduler. On the other extreme, there is 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.

As with coroutines, it is difficult to know when an execution may need to be suspended or resumed since a context change can potentially occur at any moment. Using threads to develop interruptible context-aware applications would require the developer to identify those points where an execution may be interrupted, which is almost impossible due to the unpredictable nature of context changes. The developer would have to face the burden of explicitly inserting context predicates to proceed every expression in the procedure body. Also, the developer would require explicit suspend constructs for every context condition. Even still, the developer would need to setup a custom management of suspended threads to decide when they need to be resumed. Another problem is that multiple threads may interfere with each other thus requiring the developer to deal with the problems of thread management to avoid race conditions and deadlocks. All that burden lies squarely on the shoulders of the developer.

8.4 Functional Reactive Programming

Functional reactive programming (FRP) languages [6, 20, 29, 33] provide support for automatically managing dependencies among procedures. Central to FRP is the language mechanism to declaratively express dependencies among procedures and data, and then let the language automatically re-evaluate the procedures whenever there is value change of the data they depend on. The technique we employ in the change-driven resumption of paused executions is similar to that found in FRP languages. However, the focus of FRP is mostly on efficient propagation of change and not on expressing context conditions to decide whether an execution should be executed or not. Therefore, any change of a value always triggers the execution that runs from the start to the end without interruption.

ReactiveML [25] is reactive library for Standard ML that is based on the notion of *reactive expressions*. A reactive expression is an SML expression that can be activated and suspended during its execution. The library provides constructs `activate` and `suspend` to activate and suspend an execution of a reactive expression, respectively. The main difference between ReactiveML and our approach is that ReactiveML requires the developer to explicitly express which points in the procedure body that an execution can be suspended using the `suspend` construct.

9. Conclusion

> *To advance beyond stone, we must tame fire. Only then can we forge new tools and spark a new age of invention and an explosion of new technologies. 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 [8].

The explosion of sensor-equipped devices has led us to believe that the future of mobile applications lies in *true* context-awareness. We have presented our vision of the *interruptible context-dependent executions*, which aims at spurring a programming language revolution to ease the development of *true* context-aware applications that fully exploit context information and promptly react to context changes. Our first step towards such a revolution is the Flute language, which incorporates the interruptible context-dependent executions model. As we have demonstrated, by incorporating this model, Flute:

- Facilitates the development of context-aware applications whose execution can be interrupted by context changes at any moment during the execution.
- Enables the developer to express a context condition (by means of a single context predicate) under which a context-dependent procedure is constrained to run.
- Supports a reactive dispatching mechanism that continuously takes into account of any new context changes in order to select new applicable context-dependent procedures to execute for the current context.
- Provides interruption strategies (*suspend* and *abort*) that allow the developer to specify what to do with the execution when the associated context predicate is no longer satisfied
- Provides resumption strategies (*resume* and *restart*) that allow the developer to specify what to do with the suspended execution when its associated context predicate later becomes satisfied again.
- 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.

Acknowledgments

We would like to thank Bjorn Freeman-Benson and the anonymous reviewers for their very helpful feedback and suggestions for improving the paper. We would like to thank Simon De Schutter for experimenting with a mini version of Flute in Racket. We would like to thank the members of the Software Languages Lab for the feed-

back about this work. This work is partially funded by the SAFE-IS project in the context of the Research Foundation - Flanders (FWO), the VariBru project of the ICT Impulse Programme of the Institute for the encouragement of Scientific Research and Innovation of Brussels (ISRIB), the STADiUM SBO project sponsored by the "Flemish agency for Innovation by Science and Technology" (IWT Vlaanderen), and the MobiCrant project in the context of InnovIris (the Brussels Institute for Research and Innovation).

References

[1] E. Bainomugisha, W. De Meuter, and T. 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, USA, 2009. ACM. ISBN 978-1-60558-538-3.

[2] E. Bainomugisha, J. Vallejos, E. G. Boix, P. Costanza, T. D'Hondt, and W. D. Meuter. Bringing Scheme programming to the iPhone - Experience. *Software, Practice Experience.*, 42(3):331–356, 2012.

[3] G. Birtwhistle, O. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin*. Chartwell-Bratt Ltd, 1979. ISBN 086238009X.

[4] L. Capra, W. Emmerich, and C. Mascolo. Reflective middleware solutions for context-aware applications. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, REFLECTION '01, pages 126–133, London, UK, 2001. Springer-Verlag. ISBN 3-540-42618-3.

[5] M. E. Conway. Design of a separable transition-diagram compiler. *Communications of The ACM*, 6:396–408, 1963. doi: 10.1145/366663.366704.

[6] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *European Symposium on Programming*, pages 294–308, 2006.

[7] P. Costanza. Language constructs for context-oriented programming. In *Proceedings of the Dynamic Languages Symposium*, pages 1–10. ACM Press, 2005.

[8] S. Dmitriev. Language oriented programming: The next programming paradigm. *JetBrains onBoard*, 1(2), 2004.

[9] M. Ernst, C. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, ECCOP '98, pages 186–211, London, UK, 1998. Springer-Verlag. ISBN 3-540-64737-6.

[10] D. P. Friedman and M. Wand. *Essentials of Programming Languages, 3rd Edition*. The MIT Press, 3 edition, 2008. ISBN 0262062798, 9780262062794.

[11] S. González, K. Mens, and P. 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, USA, 2007. ACM. ISBN 978-1-59593-868-8.

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005. ISBN 0321246780.

[13] R. Harper, B. F. Duba, and D. B. Macqueen. Typing first-class continuations in ml. *Journal of Functional Programming*, 3:465–484, 1993. doi: 10.1017/S095679680000085X.

[14] T. Harris, S. Marlow, S. Peyton-Jones, and M. 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, USA, 2005. ACM. ISBN 1-59593-080-9.

[15] C. T. Haynes, D. P. Friedman, and M. Wand. Obtaining coroutines with continuations. *Computer Languages, Systems Structures*, 11:143–153, 1986.

[16] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology, March-April 2008, ETH Zurich*, 7(3):125–151, 2008.

[17] M. C. Huebscher and J. A. McCann. Adaptive middleware for context-aware applications in smart-homes. In *Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing*, MPAC '04, pages 111–116, New York, USA, 2004. ACM. ISBN 1-58113-951-9.

[18] T. Kamina, T. Aotani, and H. 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, USA, 2011. ACM. ISBN 978-1-4503-0605-8.

[19] C. D. Marlin. Coroutines: A programming methodology, a language design and an implementation. *Lecture Notes in Computer Science*, 1980.

[20] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. 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, USA, 2009. ACM. ISBN 978-1-60558-766-0.

[21] T. Millstein, C. Frost, J. Ryder, and A. Warth. Expressive and modular predicate dispatch for java. *ACM Trans. Program. Lang. Syst.*, 31(2):7:1–7:54, Feb. 2009. ISSN 0164-0925.

[22] A. L. D. Moura and R. Ierusalimschy. Revisiting coroutines. *ACM Transactions on Programming Languages and Systems*, 31:1–31, 2009. doi: 10.1145/1462166.1462167.

[23] A. L. D. Moura, N. Rodriguez, and R. Ierusalimschy. Coroutines in lua. *Journal of Universal Computer Science*, 10:925, 2004.

[24] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads programming*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996. ISBN 1-56592-115-1.

[25] R. R. Pucella. Reactive programming in standard ml. In *Proceedings of the 1998 International Conference on Computer Languages*, pages 48–57. IEEE Computer Society Press, 1998.

[26] G. Salvaneschi, C. Ghezzi, and M. Pradella. Context-oriented programming: A software engineering perspective. *Journal of Systems and Software*, 85(8):1801 – 1817, 2012. ISSN 0164-1212.

[27] M. Serrano, F. Boussinot, and B. 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, USA, 2004. ACM. ISBN 1-58113-819-9.

[28] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 204–213, New York, USA, 1995. ACM. ISBN 0-89791-710-3.

[29] M. Sperber. Developing a stage lighting system from scratch. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 122–133, New York, USA, 2001. ACM. ISBN 1-58113-415-0.

[30] M. Sperber, R. k. Dybvig, M. Flatt, A. Van Straaten, R. Findler, and J. Matthews. Revised6 report on the algorithmic language scheme. *J. Funct. Program.*, 19: 1–301, August 2009. ISSN 0956-7968.

[31] E. Tanter. Contextual values. In *Proceedings of the 2008 symposium on Dynamic languages*, DLS '08, pages 3:1–3:10, New York, USA, 2008. ACM. ISBN 978-1-60558-270-2.

[32] J. Vallejos, S. González, P. Costanza, W. De Meuter, T. D'Hondt, and K. 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. ISBN 3-642-14045-9, 978-3-642-14045-7.

[33] Z. Wan and P. Hudak. Functional reactive programming from first principles. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–252, 2000.

[34] A. Warth, Y. Ohshima, T. Kaehler, and A. Kay. Worlds: controlling the scope of side effects. In *Proceedings of the 25th European conference on Object-oriented programming*, ECOOP'11, pages 179–203, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-22654-0.

A. The iFlute Platform Screenshots

The example context-aware applications we have presented have been developed and deployed on the iPad device that runs Apple's iOS 5.0.1. However, note that the iOS is just a testing platform and the context-dependent interruptible executions model and Flute are not tied to the iOS. Figure 2 shows the screenshot of

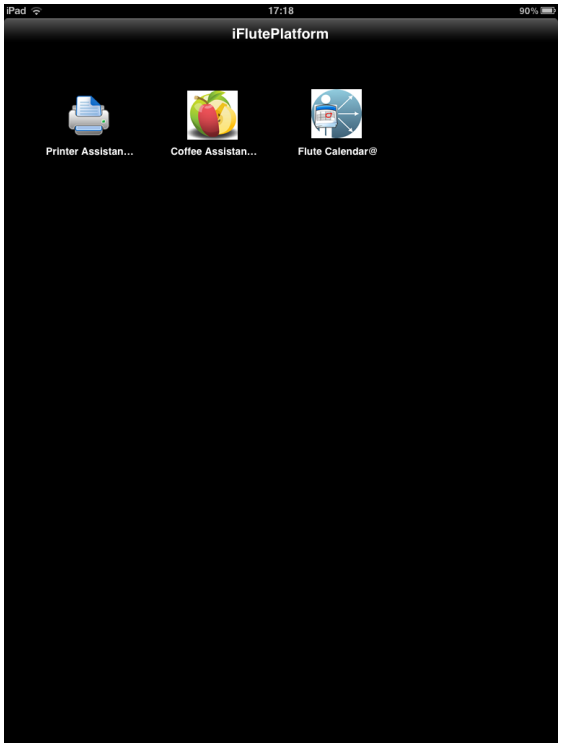

Figure 2. The iFlute platform for interruptible context-aware applications

the iFlute platform where context-aware applications can be deployed. Figures 3 and 4 show the screenshots of the context-aware calendar application and the context-aware printer assistant application running in the iFlute platform.

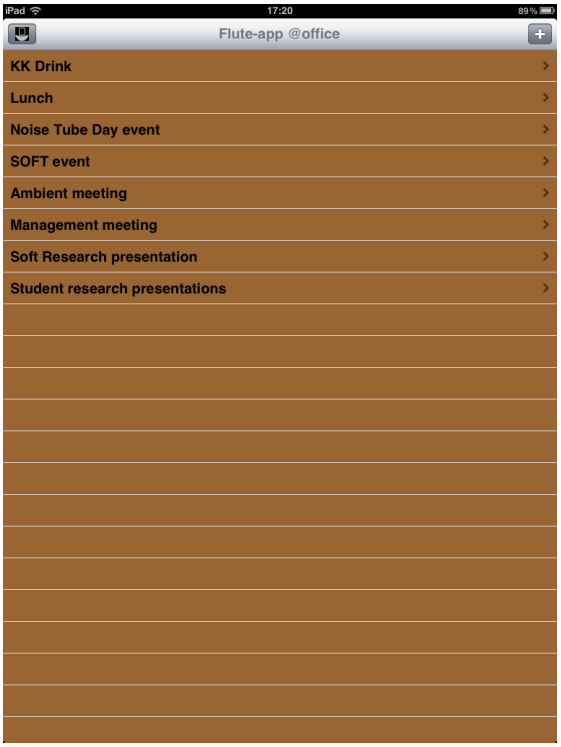

Figure 3. The context-aware calendar application executing in the public mode.

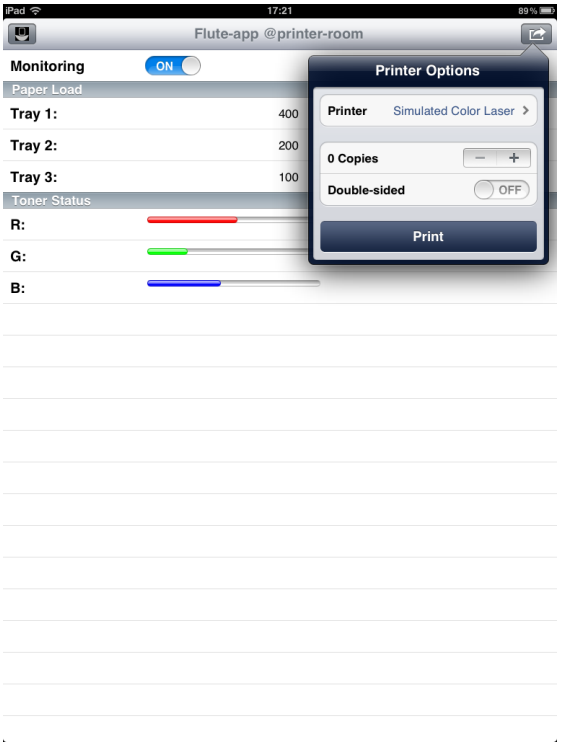

Figure 4. The context-aware printer assistant application. For this experiment, we setup a simulated Bonjour printer that supports wireless printing with the iOS.