Open Reactive Dispatch

Engineer Bainomugisha and Wolfgang De Meuter
Software Languages Lab, Vrije Universiteit Brussel, Brussels, Belgium
{ebainomu, wdmeuter}@vub.ac.be

ABSTRACT

This paper proposes the *open reactive dispatching* mechanism, where predicates are not only used for determining the applicable procedure to execute for the current situation but also for ensuring that the entire procedure execution happens under the specified predicate. Predicates operate on contextual parameters whose values change continuously (e.g., the current user's location). The dispatching process is repeated whenever the values of contextual parameters change. As a result previously unsatisfied predicates may later become satisfied and their associated procedures are selected for execution. Additionally, new procedures can be added at runtime and become part of the potential procedures that the dispatching process can select from. The open reactive dispatching mechanism is embodied in the *Flute* language, a proof-of-concept programming language that is designed for context-aware applications.

Keywords

event-based systems, predicate dispatch, open reactive dispatch

1. INTRODUCTION

Almost every program execution requires a dispatching mechanism to select the applicable procedure or method to execute for the current situation. In modern event-based systems (e.g., mobile context-aware applications) such dispatching mechanisms depend on contextual parameters (e.g., the current user's location and time of the day). Without appropriate language support, this implies for the developer to perform multiple conditional checks (e.g., by using `case` and/or `if` statements) to determine the appropriate procedure to execute for the current situation. This can negatively affect a program's composability, comprehension and maintainability (e.g., introducing new procedures or contextual parameters requires modifying multiple existing dispatching points in a program). Moreover, since such conditional checks operate on time-varying values, it implies that the applicability of the procedures to execute depends on conditions whose outcome also *changes over time*. This implies that a procedure that cannot be selected under the current context situation may *eventually* become applicable when a context change occurs at a later moment. We argue that there is a need for a dispatching mechanism that continuously selects appropriate procedures to execute for the current situation whenever context changes are observed. In order to motivate the need for such a dispatching mechanism, we introduce a scenario called *an onboard digital platform*.

1.1 Motivating Scenario: An Onboard Digital Platform

Consider an onboard digital platform that is fitted in a bus to continuously show useful information to the passengers by running a suite of applications. The applications' behaviours switch all the time as the bus drives about, as it approaches stops, as it smells other buses (e.g., by using RFID technology), when it is within a range of certain GPS-coordinates and as passengers get on or off the bus. The bus company can add or extend the existing applications and their behaviours at runtime. An example application that is deployed on the platform is the *BainoInfo* application. *BainoInfo* is an application that *continuously* shows useful information to the passengers as the bus drives about. The application automatically adapts its behaviour depending on contextual information. It shows information (e.g., name and timetable) about a bus stop as soon as the bus approaches the stop. It shows touristic information as soon as the bus approaches a museum or other touristic attractions in the neighbourhood. It shows targeted advertisements that are relevant for the current location of the bus (e.g., displaying the showing movies at the nearby cinema). Moreover, passengers can interact with the platform using their mobile phones to customise the application's behaviour (e.g., changing the display language and colour). Table 1 summarises the different behavioural variations of the *BainoInfo* application.

Context	Behaviour
Bus stop	Showing the bus stop's timetable
Touristic area	Showing touristic information
Nearby cinema	Presenting showtimes and promotions

Table 1: Behavioural variations for the *BainoInfo* application.

1.2 Scenario Analysis

The motivating example described above reveals a general pattern of issues that characterise modern event-based systems. Below we focus on three representative issues:

Need for contextual dispatch. Event-based systems such as then one described above consist of different behavioural variations for different context situations. The currently running behavioural variation depends on runtime contextual information. This requires a dispatching mechanism that takes into account contextual information (e.g., the current bus location) in order to select the appropriate behavioural variation to execute for the current situation. For instance, in the *BainoInfo* application, the behavioural variation for showing touristic information should be executed only if the bus is within range of a touristic area, while the timetable for a particular stop should be shown only if the bus is nearby that bus stop. Traditionally, such dispatching is done using explicit conditions (e.g., `if` statements). However, that approach results in convoluted code where each behavioural variation in a program has to be preceded with a contextual check.

Need for eventual and reactive dispatch. Since context changes occur continuously, it implies that the applicability of the appropriate behaviour to execute depends on contextual conditions whose outcome *changes all the time*. This implies that a behavioural variation that cannot be selected in the current context may *eventually* become applicable when a context change occurs. This necessitates a dispatching mechanism that is continuously repeated in response to context changes. For instance, in the above scenario, as the bus moves about, previously unsatisfied conditions can become satisfied. As a result, their corresponding behavioural variations should be selected for execution. Additionally, new behavioural variations may also need to be deployed to cater for new context situations (e.g., add new behaviours for new bus stops). When new behavioural variations are added, the dispatching process should automatically consider them. For developers, realising such functionality without appropriate support is not trivial.

Need for contextual-constrained executions. Another issue that needs to be addressed is that even after a behavioural variation is selected for execution, there is a need to ensure that the entire behavioural variation's execution happens in the prescribed contextual condition. For instance, in the above scenario, the bus can move away from within range of a cinema while the behavioural variation for showtimes is executing. In such case, allowing the behavioural variation to continue executing would be wrong since the showtimes behavioural variation should only be executed when the bus is within range of a cinema. Without appropriate support, this implies for developers to manually insert extra conditions in the body of a procedure or a method in order to ensure that its entire execution happens under the correct conditions. One disadvantage of implementing such concerns in that style is that the developer is required to modify multiple conditional checks in case there is a need to change a context condition (e.g., a change of location of a bus stop).

1.3 Inadequacy of Existing Dispatching Mechanisms

Predicate dispatch approaches [6, 8] alleviate some of these problems, by allowing the selection of applicable procedures to depend on arbitrary predicates. However, in predicate dispatch approaches the predicate is used only to select the applicable procedure but not for ensuring that the entire procedure execution happens under the specified condition. We characterise existing dispatching mechanisms as *closed dispatch* in the sense that they:

Dispatch and execute until completion. The current dispatching mechanisms employ some form of *dispatch and go*. The dispatching process happens only once upon each dispatching incident but when the procedure execution starts, it is up to the developer to ensure that the entire procedure execution happens in the right conditions (e.g., through repetitive conditional checks in the procedure's body). This is necessary because the initial dispatching condition may become false while the procedure execution is ongoing. Allowing a procedure execution to continue under wrong conditions may result in an incorrect application behaviour.

Provide no support for eventual applicability. Because the dispatch conditions in event-based systems depend on contextual parameters whose values change over time, it is possible that procedures that were not previously selected may become applicable at a later time. Current dispatching mechanisms only select the applicable procedure based on condition that evaluates to true at the initial dispatching time. However, once the initial dispatching process is completed the procedures that were not selected will not be executed even when their associated conditions later become true.

Have limited runtime extensibility. In dynamic software applications, it may be necessary to add new procedures at runtime even after the initial dispatching process has completed. In current dispatching mechanisms it is not possible for the procedures that are added after the initial dispatching process to be considered by the dispatcher. This implies for the developer to manually restart the dispatching process (reissue the invocation) whenever new procedures are added to an existing group of procedures to select from.

We argue that there is a need for a new dispatching mechanism that addresses the above issues. In the next section we present our proposal for the *open reactive dispatching* mechanism that aims to provide the appropriate language support for composition and dispatch in modern event-based systems.

2. OPEN REACTIVE DISPATCHING

This paper proposes a new dispatching mechanism called *open reactive dispatch*. With open reactive dispatch, *a procedure is associated with a predicate to determine its applicability* like in predicate dispatch approaches. The predicates operate on contextual parameters such as the current context of use (e.g., location, time of the day, and user preferences). A unique property of the open reactive dispatching mechanism is that the predicate is not only used for determining the applicability of a procedure but also used to ensure that *the entire procedure execution respects the predicate*. If the predicate is no longer satisfied while the procedure execution is ongoing, then the execution is promptly interrupted (aborted or suspended). The suspended execution is resumed or restarted later if its associated predicate happens to become satisfied again. Additionally, *the dispatching process is repeated whenever the values of the contextual parameters change*. When previously unsatisfied predicates – those that are associated with the procedures that were previously not selected – later become satisfied, the dispatching process is automatically repeated and new applicable procedures are selected for execution. Moreover, *new predicated procedures can be added at runtime* and are taken into account whenever the dispatching process is repeated. Below we describe the defining properties of the open reactive dispatching mechanism:

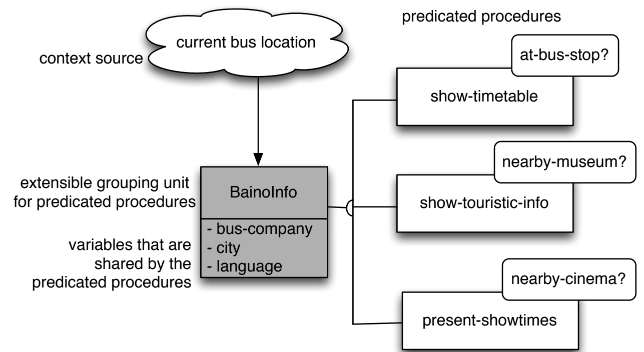

Figure 1: An extensible group of predicated procedures

Extensible composition units. Related behavioural variations (predicated procedures) are composed together under the same unit that is denoted by a name specified by the developer. The name can be referred to in other parts of a program and can be used to initiate invocation. The grouping entity also specifies any shared variables that are accessible to all predicated procedures belonging to that entity. New predicated procedures can be added to an existing entity at runtime. The addition of new predicated procedures should not require modifications existing procedure definitions. Figure 1 shows an extensible group of the predicated procedures that make up the *BainoInfo* application. *show-timetable* is associated with the *at-bus-stop?* predicate, *show-touristic-info* is associated with the predicate *nearby-museum?*, while

present-showtimes is associated with the predicate *nearby-cinema?*. The predicates operate on the context source *current bus location*, whose value changes as the bus moves about. All these predicated procedures belong to the grouping entity *BainoInfo*. The grouping entity defines variables *bus-company, city*, and *language* that are shared by all the predicated procedures.

Reactive dispatching of predicated procedures. The execution of predicated procedures is initiated by invoking the grouping entity. Invoking a grouping entity requires a dispatching mechanism to select the applicable procedure to execute. The dispatching process is broken down into the following steps:

1. The dispatching process starts by evaluating the predicates that are associated with predicated procedures belonging to the same group. A predicated procedure whose predicate evaluates to *true* is selected for execution. For instance, in the *BainoInfo* application (cf. Figure 1), if the bus happens to be nearby a bus a stop, the predicate *at-bus-stop?* will evaluate to true hence the *show-timetable* procedure is selected for execution.
2. During the execution of the selected predicated procedure, its associated predicate is re-evaluated at every step in the procedure body in order to ensure that the procedure only executes in the correct situation. If the predicate is no longer satisfied, then the procedure execution is interrupted – suspended or aborted. The kind of interruption is specified by the developer. The suspended procedure execution can be resumed or restarted if its associated predicate happens to become satisfied again. Like with the interruption, the kind of resumption is specified by the developer. For instance, in the *BainoInfo* application, the *show-touristic-info* procedure should run only if the bus is nearby a museum. So, if the bus moves out of range while the *show-touristic-info* procedure's execution is ongoing, the execution is immediately interrupted. If the bus later returns to the location the execution can be resumed or restarted.
3. The dispatching process is repeated whenever the values of contextual sources change. This implies that previously unsatisfied predicates may become satisfied and as result their associated procedures will be selected for execution. Additionally, any new predicated procedures that are added (after the previous dispatching process) are also considered whenever the dispatching process is repeated. For instance, in the *BainoInfo* application, suppose that the bus is initially nearby a museum. This implies that the *nearby-museum?* predicate evaluates to true while the predicates *at-bus-stop?* and *nearby-cinema?* evaluate to false. However, if the bus later approaches a bus stop, the dispatching process is automatically repeated and the *show-timetable* procedure will be selected for execution since its associated predicate – *at-bus-stop?* – will now evaluate to true.

Reactive scope management. Because the execution of procedures can be interrupted at any moment, it is desirable to ensure that they resume in a consistent execution environment. Different predicated procedures may be manipulating the same data. As such the currently interrupted executions may be resumed when shared data has been modified by other executions. For instance, in the *BainoInfo* application (cf. Figure 1), if the *language* variable is modified during the execution of the *show-touristic-info* procedure, its changes may be visible by other procedures that share that variable. It is therefore necessary to ensure that modifications to the shared data do not lead to inconsistencies. To this end, each predicated procedure is associated with a developer specified scope management strategy for controlling the visibility of changes to the variables that are shared by several predicated procedures. The scope management strategies are: (i) *Immediate visibility* – where state changes to the variables that are shared among predicated procedures are immediately visible by other executions that share that variable, (ii) *Deferred visibility* – where changes remain local to the execution on interruption but become visible to other executions on completion of the procedure execution, and (iii) *Isolated visibility* – where changes are restricted to the procedure that made the changes and they are not visible by other procedures.

2.1 Supporting Open Reactive Dispatching

The open reactive dispatching mechanism is embodied in the *Flute* language [2], a proof-of-concept programming language that is designed for context-aware applications. Flute is built as a meta-interpreter in our Scheme implementation called iScheme [1], which runs on the iOS platform. The syntax of the Flute language is essentially an extension of that of the Scheme language [5]. In the remainder of this section, we discuss the language support for open reactive dispatching in Flute. We will do so by using the motivating example of the *onboard digital platform* that we introduced in Section 1.1.

2.1.1 Defining extensible composition units (modals)

Flute provides a composition abstraction, a *modal* that allows the developer to define a composition unit that groups together related predicated procedures. A modal specifies the contextual sources on which the predicates operate. It also allows the developer to define variables that are shared among the predicated procedures belonging to the same modal. A modal is extensible in the sense that new predicated procedures can be added to it at runtime without requiring modification of the modal definition and existing predicated procedures.

For instance, when developing the *BainoInfo* application, we require a modal that groups together all its different behavioural variations (predicated procedures). Such a modal is defined as follows:

Listing 1: Defining modals

```
1 ;defining a contextual source
2 (define bus-location (ctx-event))
3
4 ;defining a modal
5 (define bainoinfo (modal (bus-location)
6    (define bus-company "BainomugiStar")
7    (define city        "Kampala")
8    (define language    "EN")))
```

The current location of the bus is represented as a contextual source. A contextual source is created using the `ctx-event` construct. In the above code excerpt the resulting contextual source for the current bus' location is bound to `bus-location`. A modal is created using the `modal` construct, which takes as arguments a list of contextual sources and an optional definition of shared variables. In the above code listing, `bainoinfo` is a modal for grouping together the behavioural variations of the *BainoInfo* application. It specifies the contextual source as `bus-location` and defines shared variables `bus-company`, `city`, and `language` that are accessible to all predicated procedures belonging to that modal. In the next section, we discuss the support for defining predicated procedures.

2.1.2 Defining predicated procedures (modes)

Behavioural variations (predicated procedures) are represented as *modes*. Each mode specifies the modal it belongs to. In addition, each mode is associated with a predicate that specifies its applicability. Predicates operate on the contextual sources that are specified as part of the modal definition that the mode belongs to. New modes can be added to an existing modal at runtime.

An application is composed of different modes for different behaviours. Each mode represents the behaviour that corresponds to a particular context situation. For instance, the *BainoInfo* application consists of: a

mode for showing touristic information when the bus is within range of a touristic area, and a mode for showing the timetable for a particular stop when the bus is nearby that bus stop. Such modes can be defined as follows:

Listing 2: Defining modes.

```
1  ;a mode for showing bus stop's timetable
2  (mode (bainoinfo)
3    (at-bus-stop? bus-location)
4    (suspend resume deferred)
5    (lambda ()
6      (show bus-company)
7      (show city)
8      (show (get-stop-name))
9      (show (get-timetable))
10     ...))
11
12 ;a mode for showing toursitic information
13 (mode (bainoinfo)
14   (nearby-museum? bus-location)
15   (suspend resume isolated)
16   (lambda ()
17     (show bus-company)
18     (show city)
19     (set! language "NL")
20     (show (get-museum-name))
21     (show (museum-attractions))
22     ...))
```

A mode is created using the `mode` construct. It takes as argument the modal it belongs to, a predicate expression, a list of interruption, resumption and scope management strategies, and a procedure that is created using the `lambda` special form. Modes belonging the same modal have the same list of parameters. In this example the two modes have no parameters. The mode for showing the timetable for bus stops is specified with the `at-bus-stop?` predicate, while that for display the touristic information is specified with the `nearby-museum?`. Both modes belong to the `bainoinfo` modal and operate on the `bus-location` context source that is specified as part of the `bainoinfo` definition. Additionally, each mode is specified with an interruption strategy – that specifies what do when the predicate is no longer satisfied during the execution of the mode, a resumption strategy – that specifies what to do when the predicate becomes satisfied again, and scope management strategy – that specifies the scope of changes made to the variables that are shared among all modes. Note that both modes have access to the variables `bus-company`, `city`, and `language` that are defined as part of the `bainoinfo` modal. In next section, we discuss the dispatching mechanism for selecting the appropriate mode to execute.

2.1.3 Contextual and reactive dispatching for modes

A modal is invoked as `(modal-name <arguments>)`. For instance, the `bainoinfo` modal is invoked as `(bainoinfo)` since its modes take no arguments. Modes are not invoked directly[1]. When a modal is invoked, the dispatching process is initiated to select the applicable mode to execute for the current situation. The dispatcher evaluates all the predicates that are associated with the modes belonging to that modal. The mode whose predicate evaluates to *true* is selected for execution. However, unlike traditional dispatching mechanisms where the dispatching process happens only once upon each dispatching incident, in Flute the dispatching process is repeated when context changes are observed. Since context changes typically occur continuously, it is possible that some 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 contextual sources that may affect the contextual predicates. The dispatching process continuously monitors context changes and is automatically triggered again whenever context sources receive new values. This means that modes that were not previously selected for execution may be selected later if their associated predicates eventually evaluate to true.

[1]When a mode is bound to a name it is possible to invoke it directly. However in that case the initial dispatching process of selecting the applicable procedure is not necessary since there the name is associated with only one procedure implementation.

2.1.4 Interruptible and resumable execution of modes

The predicate that is associated with a mode is not only used for dispatching but also for ensuring that the entire mode's execution happens in the prescribed condition. Throughout the evaluation of the body expressions of a mode, the predicate that is associated with that mode is re-evaluated. If the predicate evaluates to false then the execution is interrupted (i.e., suspended or aborted) depending on the developer specified interruption strategy. If the predicate later evaluates to true again, then the previously interrupted execution is resumed or restarted depending on the developer specified resumption strategy. For instance, in Listing 2, the mode for showing the bus timetable is specified with the `suspend` strategy and the `resume` resumption strategy. Additionally, each mode has a scoping strategy for controlling the visibility of state changes. For instance, the mode for showing touristic information is specified with `isolated` strategy. This means that the state change `(set! language "NL")` is not visible to other modes that share that state.

2.1.5 Runtime addition of modes to modals

Note that it is possible to add new modes to an existing modal at runtime and when they are added they automatically become part of the potential modes that the dispatching process can select from for execution – even if the modal was already invoked. For instance, the mode for the displaying the currently showing movies when the bus is within range of a cinema can be added to the `bainoinfo` modal as follows.

Listing 3: Adding a new mode at runtime.

```
1 (mode (bainoinfo)
2   (nearby-cinema? bus-location)
3   (abort restart deferred)
4   (lambda ()
5     (show bus-company)
6     (show city)
7     (show (get-cinema-name))
8     (show (get-showing-movies))
9     ....))
```

Listing 3 shows how to extend the `bainoinfo` modal with a new mode. The mode is specified with the `nearby-cinema?` predicate. Additionally, it is specified with the `abort` interruption strategy, `restart` resumption strategy, and `deferred` scope management strategy. Adding a new mode does not require re-invoking the modal and the new mode has access to the shared variables `bus-company`, `city`, and `language` of the `bainoinfo` modal.

2.1.6 Scoping semantics for predicates and modes

Flute adheres to the *lexical scoping* semantics of the Scheme language. However, the support for open reactive dispatching requires a change in the variable lookup semantics. This is particularly important for the evaluation of predicates and modes that are added at runtime to modals. All modes belonging to the same modal have access the shared variables and contextual sources that are specified in the modal definition. Variable lookup for a predicate evaluation happens as follows:

1. To evaluate a predicate, the lookup of variables starts from the environment of the modal, which includes contextual sources and the shared variables that are defined in the modal.
2. If the variable is not found in the modal environment, the lookup proceeds to the enclosing environment of the mode.

Variable lookup for the mode evaluation happens 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 environment of the modal in which it belongs.
3. If the variable is not found in the modal environment, the lookup proceeds to the enclosing environment of the mode.

2.2 Discussion and Limitations

The previous sections have presented the support for the *open reactive dispatching* mechanism in the Flute language and illustrated it by implementing the *BainoInfo* application that runs different modes depending on the current situation. A modal serves as a composition unit that allows developers to add new modes at runtime without requiring modification to existing modes. The predicate associated with each mode is not only used to select the right mode to execute but also for ensuring that the entire mode execution is constrained to the correct conditions. Additionally the dispatching process is repeated when context changes are observed. Realising such functionality using existing predicate dispatch approaches implies for the developer to insert dispatching points in the procedure body, which can lead to negative effects on the program composability. Moreover, in existing predicate dispatching approaches the dispatch process happens once upon each dispatching incident and any predicates that later become true are not taken into account. We further discuss the related approaches in Section 3.

The current Flute implementation for supporting the open reactive dispatching mechanism is only a proof-of-concept prototype and there remain limitations that we discuss below.

Handling of return values. The fact the dispatching process may be repeated and several modes may be invoked there is need to specify how to handle multiple return values. For instance, it may be desirable to provide options to the developer to select from or merge the return values.

Ambiguous contextual predicates. Because predicates operate on contextual parameters whose values change over time, it is possible that more than one predicate may be satisfied at the same time. A possible solution to this problem is to allow the developer specify the priority order of predicates [8].

3. RELATED WORK

Predicate dispatch approaches [6, 8] employ a dispatching mechanism where the applicability of a procedure or a method depends on arbitrary predicates. Like in our approach each procedure is associated with a predicate that specifies the condition under which the procedure is allowed to execute. This eliminates the need for explicit conditional checks for selecting the procedure to execute for the current situation. However, those approaches do not provide support to ensure that the entire procedure execution allowed to execute in the correct condition. Also, dispatching process in the predicate dispatch approaches happens only once upon each dispatching incident. This means that if certain predicates later become true, their associated procedures are not considered for execution unless the procedures are invoked again. *Context-oriented programming* (COP) approaches [3] provide support for contextual dispatch. However, in COP languages, the dispatching of applicable behavioural variations happens only once upon each dispatching incident. *Guards* [4] are synchronisation mechanisms that are used to guard a method such that it only starts executing under certain conditions. Like in our approach they provide a construct to enable the developer associate a precondition with a method that specifies the condition under which the execution is allowed to begin. However, with guards, the condition that is associated with a method is only used as a precondition for a method invocation and not for ensuring that the entire method execution occurs in the specified condition. *Coroutines* [7] also provide support to express interruptions (through suspension) and resumptions. However, the developer has to *explicitly* transfer control using the `yield` construct at certain points in the procedure body. In event-based systems, such control transfers have to happen based on context conditions. As a consequence, the developer has to insert repetitive control transfer constructs and guard each one of those with a context condition.

4. CONCLUSION

This paper has presented the *open reactive dispatch* as a dispatching mechanism that improves the composition and eases the development of modern event-based systems such as context-aware applications. It uses the predicate not only to select the applicable procedure but also to ensure that entire procedure execution is only allowed to proceed when the specified condition is satisfied. Predicates operate on contextual parameters whose values change over time. The dispatching is repeated whenever the values of contextual parameters change and thus previously unsatisfied predicates may become satisfied and their associated procedures are selected for execution. The open reactive dispatching mechanism has been embodied in the Flute programming language. Flute provides composition abstractions *modals* and *modes* that allow developers to represent behavioural variations as predicated procedures and group them together in extensible composition units. For future work we plan to investigation techniques for addressing the limitations discussed in Section 2.2.

5. ACKNOWLEDGMENTS

This work was partially funded by the SAFE-IS project in the context of the Research Foundation - Flanders (FWO).

6. REFERENCES

[1] 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.

[2] E. Bainomugisha, J. Vallejos, C. D. Roover, A. Lombide Carreton, and W. D. 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*, ONWARD '12, New York, NY, USA, 2012. ACM. (To appear).

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

[4] D. Lea. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[5] J. Matthews and R. b. Findler. An operational semantics for Scheme. *J. Funct. Program.*, 18(1):47–86, Jan. 2008.

[6] T. 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.

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

[8] 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.