Event-based Modularization

How Emergent Behavioral Patterns Must Be Modularized?

Somayeh Malakuti [*]
Software Technology group
Technical University of Dresden, Germany
somayeh.malakuti@tu-dresden.de

Mehmet Aksit
Software Technology group
University of Twente, the Netherlands
m.aksit@utwente.nl

Abstract

Nowadays, detecting emergent behavioral patterns in the environment, representing and manipulating them become the main focus of many software systems such as traffic monitoring systems, runtime verification techniques and self-adaptive systems. In this paper, we discuss the need for dedicated linguistic constructs to modularly represent emergent behavioral patterns and their lifetime semantics. We explain the shortcomings of current languages with this regard. Inspired from the evolution of procedural languages to object-oriented and aspect-oriented languages, we explain the concept of **event-based modularization**, which can be regarded as the successor of the aspect-oriented modularization for representing emergent behavioral patterns and their lifetime semantics. We report on our work on **event modules** and their successor **gummy modules**, which facilitate representing behavioral patterns as a holistic module that encapsulates its lifetime semantics.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Modules, packages

General Terms Languages, Design

Keywords event modules; gummy modules; emergent behavior

1. Introduction

There are various kinds of software systems that deal with detecting emergent behavioral patterns (in short behavioral patterns) in environment, representing them in the system and facilitating the manipulation of the behavior. Among others, behavioral patterns and their lifetime semantics can be regarded as the concerns of interest, which must be represented as first-class entities in the system. To this aim, one has to deal with the following questions: a) are current programming languages, modularization and composition mechanisms suitable enough to represent behavioral patterns and their lifetime semantics, as desired? b) how can we discover and introduce new language mechanisms that may offer better software qualities?

This paper summarizes our study on the shortcomings of the current languages for modular implementation of behavioral patterns. We explain the concept of **event-based modularization**, which has been inspired from the evolution of procedural modularization to object-oriented (OO) and aspect-oriented (AO) modularization mechanisms. This concept can be regarded as the successor of AO for effectively modularizing behavioral patterns and their lifetime semantics. As an implementation of this concept, we explain **event modules** and their successor **gummy modules**, which facilitate representing behavioral patterns as a holistic module that encapsulates its lifetime semantics.

This paper is organized as follows. Section 2 outlines a set of requirements to modularize behavioral patterns and their lifetime semantics; Section 3 evaluates a representative set of languages; Section 4 discusses the concept of event-based modularization along with event modules and gummy modules. Finally, section 5 outlines our future work.

2. Representing Emergent Behavioral Patterns

There are various kinds of software systems that deal with detecting the emergence of certain behavioral patterns in environment, representing them in the system and facilitating the manipulation of the behavior. Runtime verification techniques, self-adaptive software systems, traffic monitoring software systems, stock market analysis systems, and flight control systems are examples.

Figure 1 shows an example traffic monitoring system, which consists of a *physical* and a *cyber* part. The *road segments* and the *sensors* embedded in the road segments form the physical part. In the cyber part, traffic congestion in a road segment is regarded as a behavioral pattern of interest. There are software entities, which receive information from the sensors, reason about the density of the traffic flow in the corresponding road segments, and if the density is above a certain threshold, they infer that there is traffic congestion in the corresponding road segments. The traffic congestion is represented as a runtime entity (e.g. an object) in the system, which maintains the information of interest, such as average speed and density of the traffic congestion. This entity can be accessed by other application objects, for example to display and/or analyze the information. Somewhere during the execution of the system, traffic congestion may disappear in a road segment; implying that the corresponding entity must be destroyed accordingly.

Software systems that deal with behavioral patterns are in general complex, exhibit dynamic structure, and due to high development costs, they must be implemented to be long-lived. Therefore, it is necessary to properly implement and modularize the relevant

[*] The author is supported by the German Research Foundation (DFG) in the Collaborative Research Center 912 ”Highly Adaptive Energy-Efficient Computing”.

FOAL '14, April 22, 2014, Lugano, Switzerland.
Copyright © 2014 ACM 978-1-4503-2798-5/14/04…$15.00.
http://dx.doi.org/10.1145/2588548.2588549

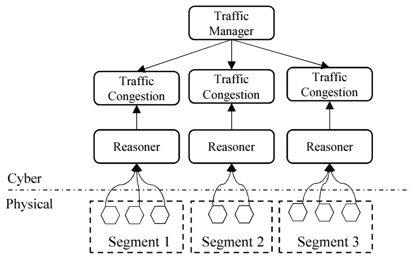

Figure 1. An example traffic management system

concerns. To this aim, we claim that a language must fulfill the following requirements:

- ***Acquisition of data:*** First of all, the language must provide means to select the relevant data from the environment for the purpose of determining the appearance and disappearance of the behavioral patterns of interest. The kinds of data cannot be fixed in general, and is application specific. For example, in the traffic management system, the data that is provided by the sensors can be traffic flow information, pollution information, etc. This implies that the language must be open-ended with respect to the set of supported data.
- ***Detection of the appearance and disappearance of behavioral patterns:*** The language must provide suitable constructs to express the semantics for appearance and disappearance of behavioral patterns. Such semantics usually express a correlation among the selected data. For example, one may define the following rules for traffic management systems: if the density of traffic in a road segment goes beyond the threshold *D* in the time duration of *T*, it is inferred that traffic congestion has appeared in the road segment. Likewise, if the density drops below this threshold, it is inferred that the traffic congestion has disappeared.
  As it has extensively being studied in the runtime verification community [10, 11], different formalisms may be adopted to express appearance and disappearance semantics. This implies that to increase the usability of the language, it must be also open-ended with respect to the set of supported formalisms.
- ***Modular representation of behavioral patterns:*** Software systems dealing with behavioral patterns possibly consist of many different concerns. To have better software qualities, it is generally assumed that concerns must be modularized. A well-established definition of a module is: it offers an interface and its implementation is encapsulated. A behavioral pattern itself may be regarded as a concern of interest, which must be modularized. For example traffic congestion may be modularized as an object, which encapsulates information about the density and length of the traffic congestion, and provides various methods such as *display*, *log* and *regulate* to access and manipulate this information.
  In the modularization of a behavioral pattern, its lifetime semantics, which consists of the following four sub-concerns, must also be taken into account: a) appearance semantics, b) disappearance semantics, c) utilization semantics, and d) the synchronization semantics among these. As it is explained before, the appearance and disappearance semantics express the condition under which a behavioral pattern is regarded as appeared and disappeared in the environment, respectively. The utilization semantics express the set of state information and operations that are defined for the behavioral pattern. The rules to synchronize these must also be defined; for example, an object representing a behavioral pattern can only be utilized between the time interval that the behavioral pattern has appeared and has not disappeared. Likewise, the object can only be destroyed after it is concluded that the behavioral pattern has disappeared.
  We claim that the module abstractions of a language must be expressive enough to modularize these sub-concerns if needed, and must be expressive enough to integrate them with each other so that a modular representation of the corresponding behavioral pattern can be achieved.

3. Implementation Mechanisms

There are several alternative programming languages and implementation mechanisms that can be adopted to express behavioral patterns and their related sub-concerns. We have extensively studied the shortcomings of the current languages for this matter [7–10]. In this section, we summarize our observations.

3.1 Object-Oriented (OO) Languages

A straightforward approach is to adopt objects to represent the behavioral patterns of interest and their lifetime semantics. To improve the modularity of implementations, various design patterns [3] can be adopted for this matter.

For example, the Observer pattern can be adopted to implement the functionality for selecting the data of interest from environment. Here, the environmental entities that provide the data get the role of *subject*, which through invoking the method *notify* provide the data to the objects with the role of *observer*. Individual observers may be provided to gather individual kinds of data. The functionality to reason about the correlation of data, which is possibly gathered by individual observers, can be defined in separate objects interacting with the observers. The Factory pattern can be adopted to construct an object representing a behavioral pattern. The State pattern can be adopted to maintain the current state of the behavioral pattern; i.e. appeared, disappeared, or being utilized.

There are several well-known problems associated with the OO languages and design patterns. First of all, programmers must have extensive knowledge of each design pattern and its constraints. Second, the implementations can easily become complex and hard to comprehend, because each pattern requires its own classes, its class hierarchies with specific methods, and its constraints. One may need to sacrifice the constraints of patterns and/or has to introduce new patterns to combine existing patterns such that the constraints of the patterns are respected [5].

Advanced OO languages usually offer an event-delegate mechanism, such as the one offered by C#, to facilitate implementing event-based applications. Such event-delegate mechanisms are, however, ad-hoc patches to OO languages, which come with the same shortcomings discussed for the Observer-based implementation. Due to the space, we refer the interested readers to [7, 9] for a detailed discussion over these shortcomings. There are more advanced language extensions, such as the ones provided by Esper [2]; it offers dedicates means to define complex event processing semantics as an extension to SQL. The SQL-like language may not be expressive enough to define various appearance and disappearance semantics of behavioral patterns as it is extensively studied in the runtime verification community [10, 11].

3.2 Aspect-Oriented (AO) Languages

AO languages offer certain features that can reduce and/or eliminate the need for adopting design patterns in modularizing behavioral patterns. In an AO implementation of behavioral patterns, join points can be regarded as means to represent state changes of interest in the environment; the data of interest can be provided through join point contexts. Base objects in which join points are activated can be regarded as environmental entities. Pointcut designators are means to query the join points of interest; advice code provide the functionality to react to the activated joint points. In most AO languages, aspects are means to modularize a set of correlated pointcuts and advice, as well as local variables and methods. Therefore, one may consider modularizing a behavioral pattern as an aspect. However, the limitations of the current AO languages prevent achieving a proper modularization as explained in the following.

The set of supported joint points and contextual data are defined by the join point model of the adopted AO language. Most AO languages such as AspectJ, CaesarJ [12], and Compose* [1] support a fixed join point model. There are various proposals such as [4] to support programmable join point models, which are mainly limited to Java as the base language. As we studied in [8, 10], supporting a single base language reduces the modularity of implementations when data is provided by various sources, for example multi-language base software.

In the AO languages that support pointcut-based instantiation of aspects (e.g. AspectJ-like languages), the appearance semantics of behavioral patterns can be expressed through pointcut designators and the instantiation strategy of aspects, if they are expressive enough for this matter. Otherwise, workarounds must be provided via advice code and helper methods and objects, which complicate the implementations. Several proposals exist to offer more expression power via stateful pointcuts [14]. These proposals fix the language in which pointcuts can be expressed. However, as it is studied in the runtime verification community [10, 11], usually a diverse set of languages, such as regular expression, state machines, and temporal logics are necessary to define the appearance semantics of behavioral patterns.

In the AO languages that support pointcut-based instantiation of aspects, the disappearance semantics of behavioral patterns cannot be directly expressed because the lifetime of aspects usually depends on the lifetime of the corresponding base objects. Therefore, workarounds must be provided to mark an aspect as destroyed to emulate the disappearance of behavioral patterns. In the languages that support explicit construction and deployment of aspects, such as CaesarJ [12], the same problems as the ones in the OO languages can be observed; multiple design patterns must be adopted to implement the appearance, disappearance and synchronization semantics of behavioral patterns.

There are various languages/extensions, such as Ptolemy [13] and EventCJ [6], that adopt features of AO languages to facilitate implementing event-based applications. Although each of these languages have their own advantages, they inherit many shortcomings of AO languages. The previously discussed limitations of AO languages representing diverse kinds of data that must be selected from objects implemented in different languages, expressing complex event selection semantics via pointcut designators, expressing complex appearance and disappearance semantics also exist in these languages.

4. Event-based Modularization

Since existing languages significantly fall short in implementing event processing applications we face the question: *how can we discover and introduce new language mechanisms that may offer better software qualities than the constructs of the current languages?*

To answer this question, we try to learn from the evolution of procedural languages to OO and AO languages, which is shown in Figure 2. Basically, a procedural language is based on a calling procedure, an explicit call with zero or more call parameters, a callee procedure with an implementation code. The callee procedure executes the call, and it may invoke on other procedures; this form of communication is known as the client/server communication.

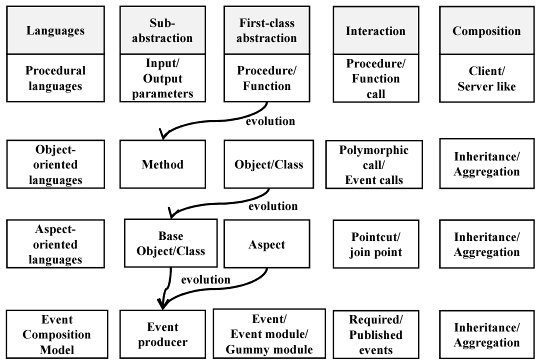

Figure 2. Evolution of modularization mechanisms

The OO paradigm has been introduced by grouping a set of procedures (methods) together under the concept of object. These methods can be explicitly invoked, or if the language offer an event-delegate mechanism, they can be invoked implicitly via event announcement. It has been claimed that object abstraction reduces complexity and enhance reuse since they may better correspond to the reusable 'real-world entities' in the problem domain. Further, objects and calls can be typed, and reused using the class and class inheritance concepts and calls can be made polymorphic. These features are claimed to reduce complexity and increased reuse and evolvability, because tightly coupled procedures are now grouped under the same linguistic abstraction and loosely coupled procedures are related through polymorphic calls and inheritance.

Current AO programing languages extend the OO model through implicit call mechanisms; objects do not need to refer themselves through explicit names but through quantification predicates. This provides a more reusable and flexible coupling among the caller and callee procedures, since the bindings are more associative than direct.

Current programming languages lack abstraction and reuse mechanisms for implementing and modularizing behavioral patterns as desired; they seem to be defined more at a procedural level. Inspired from the evolution history of procedural, OO and AO programming languages, we introduced the concept of *event-based modularization* via Event Composition Model [10]. Event Composition Model offers **events**, **event modules** and **gummy modules** as the successor of aspects to effectively modularize behavioral patterns.

At a high level of abstraction, Event Composition Model considers the environment as a set of **events**, which may be published by the objects and aspects that exist in an application, and/or by external entities such as OS and hardware derivers. Events are typed entities; an event type defines a set of attributes for the events. To overcome the problems of fixed join point models, new kinds of event types, attributes and events can be defined according to applications demands.

In [10], we introduced **event modules**, which is implemented by the EventReactor language, as a means to modularize a group of related events and the reactions to them. As Figure 3 shows, an event module has a **required interface**, an implementation that is termed as **reactor**, and a **provided interface**. The required interface of an event module, which is analogous to pointcut designators, queries the set of events of interest to which the event module must react; the language in which the queries can be expressed is not fixed.

Reactors, which are analogous to advice code, provide the functionality to process the events specified in the required interface of the event module, and to publish the so-called **reactor events** to the environment. These events form the provided interface of the event

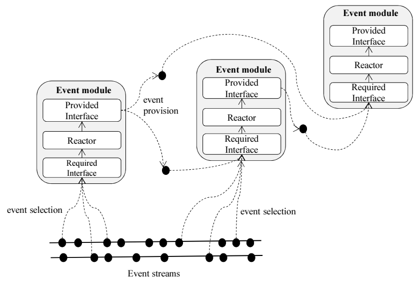

Figure 3. Event modules

module, which is analogous to the set of joint points that can be designated in an aspect module. As for other elements, the language for expressing reactors is not fixed, and various general-purpose and/or domain-specific languages can be adopted for this matter.

The events published by an event module can be selected further by other event modules. This facilitates the composition of event modules with each other, and modularly expressing composition constraints as event modules in their desired language.

Although event modules significantly improve upon current OO and AO modularization mechanisms to implement event processing applications [8–10], they come with some shortcomings. First, the instantiation and destruction strategies of event modules are largely similar to the ones offered by the AO languages with pointcut-based instantiation strategy. Consequently, workarounds must be provided to express complex appearance disappearance semantics of behavioral patterns with the price of reduced modularity. Moreover, event module instances cannot be accessed by other objects and be utilized as first-class entities. This limits the possibility to define operations and invoke them on event module instances.

To improve the modular representation of behavioral patterns, we are investigating **gummy modules** as the successor of event modules. As Figure 4 shows, a gummy module is also defined in terms of **required interface**, **reactor** and **provided interface**, which are bound to each other. These elements can be either primitive or composite. A primitive required interface and provided interface represents an event or data that is selected and provided by a gummy module, respectively. The primitive reactor is an executable program to process these events. The composite specializations of required interface and provided interface are gummy modules too; this facilitates defining gummy modules with nested structures. Gummy modules have two special composite required interfaces termed as **appearance** and **disappearance**, which encapsulate the appearance and disappearance semantics of behavioral patterns, respectively. Adopted from event modules, the language in which these semantics can be expressed is not fixed; various DSLs and/or GPLs can be adopted.

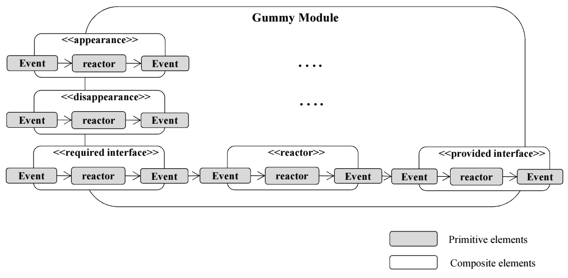

Figure 4. Gummy modules

A behavioral pattern can be represented as one holistic gummy module, which encloses sub-gummy modules that express its appearance, disappearance and utilization semantics. The enclosing gummy module may define local variables that can be accessed by its sub-gummy modules.

The synchronization semantics is currently fixed by the language. At runtime before an instance of a gummy module is constructed, its **appearance** interface is the only active interface. This interface selects the events of interest from the environment, reasons about the runtime condition, and concludes whether an instance of the gummy module must be constructed. If so, it publishes a specific event to the runtime environment of the language to construct an instance of the gummy module. After instantiation, the **disappearance** interface as well as other required interfaces of that instance become active. Likewise, the **disappearance** interface concludes whether the instance must be destroyed. While an instance of a gummy module is alive, the other required interfaces select the events of interest from the environment, and forward them inside the gummy module to the corresponding reactors to process them. As a result of the processing, new events may be published to the environment. The semantics for publishing these events are defined within the provided interfaces of the gummy modules, which are bound to the reactors.

Listing 1 shows a simplified gummy module for representing traffic congestion. Here, we assume that road segments publish events of the type `SegmentEvent`, which carry information about the density of traffic flow. The details of defining and publishing events can be found in [10].

Lines 1 to 13 define the gummy module `TFAppearance`, which encapsulates the semantics for the appearance of traffic congestion in a road segment. This module receives an event of the type `SegmentEvent` and a list of accepted publishers for the event via the required interfaces `event` and `publishers`, respectively. The module publishes an event of the predefined type `Construct` via its provided interface `instantiate`. As its implementation in line 5, the local variable `densities` is defined to accumulate information about the density of traffic flow in the road segment.

Lines 6–13 define the program `checkAppearance`, which first checks whether the publisher of the input event exists in `publishers`; then it accumulates information about the density of the traffic flow for `T` seconds in the local variable `densities`. After this time is passed, it computes the average density of the traffic flow, and if it is above the threshold `D`, it publishes the event `instantiate` to inform the runtime environment that an instance of the enclosing gummy module must be constructed to represent the traffic congestion in the program. Afterwards, it updates the local variable `density` in its enclosing gummy module.

The variables proceeded with the character `?` are pseudo variables with special meanings. The predefined variable `?input` refers to the event whose occurrence causes the program be executed. The predefined variable `?self` refers to the current gummy module or any of its enclosing ones. Within the implementation, it is possible to refer to the variables defined in the required and/or provided interface of the module, providing that their name is proceeded with the character `?`. `TMSTimer` and `TMSMath` are helper Java classes that implement the functionality to compute the elapsed time and average density, respectively.

The binding expression in line 14 specifies that whenever there is an event bound to the required interface `event`, the program `checkAppearance` must be executed.

Lines 16–24 likewise define `TFDisappearance`, which encapsulates the semantics for inferring the disappearance of traffic congestion, and publishes the event `destroy` of the predefined type `Destruct` to inform the runtime environment that the gummy module representing the corresponding traffic congestion be destroyed.

Lines 25–30 define the gummy module `TFUtilization`, which defines the functionality to log traffic congestion information. To this aim, it receives an event of the type `LogEvent`, which triggers the program `log`. This program prints the value of the variable `density` defined in the gummy module enclosing `TFUtilization`.

Lines 31–39 define the gummy module `TrafficCongestion`, which modularly represent traffic congestion and its lifetime semantics. This module is composed of the sub-gummy modules `appearance`, `disappearance` and `utilization` as its required interfaces. The gummy module publishes the events `construction` and `destruction` to inform the runtime environment that its instances must be constructed and destructed, respectively. Within the part `reactors`, the variables that are shared among the sub-gummy modules are defined. In lines 37–38, the provided interface of the sub-gummy modules `appearance` and `disappearance` are bound to the corresponding provided interface of `TrafficCongestion`, so that the events can become visible outside the sub-gummy modules.

```
 1 gummymodule TFAppearance is Appearance{
 2  required interfaces{SegmentEvent event; List<Long> publishers;}
 3  provided interfaces{Construct instantiate;}
 4  reactors{
 5    variables {ArrayList<Long> densities = new ArrayList<Long>();}
 6    program checkAppearance{
 7     if ( TMSList.contains(?publishers, ?input.publisher)){
 8        if (TMSTimer.computeElapsedTime() < ?self.T)
 9         densities.add (?input.getAttribute("density"));
10        else { Long avg = TMSMath.computeAverage(densities);
11              if (avg > ?self.D) { publish(?instantiate); ?self.density = avg; }
12        }
13    } }}
14  bindings{bind(event, checkAppearance);}
15 }
16 gummymodule TFDisappearance is Disappearance {
17  required interfaces{SegmentEvent event; List<Long> publishers;}
18  provided interfaces{Destruct destroy;}
19  reactors{
20    variables {ArrayList<Long> densities = new ArrayList<Long>();}
21    program checkDisappearance{ ... }
22  }
23   bindings{bind(event, checkDisappearance);}
24 }
25 gummymodule TFUtilization{
26  required interfaces{LogEvent event;}
27  provided interfaces{}
28  reactors{ program log(){ System.out.println(?self.density); }}
29  bindings { bind (event, log);}
30 }
31 gummymodule TrafficCongestion {
32  required interfaces{
33   TFAppearance appearance; TFDisappearance disappearance;TFUtilization utilization;}
34  provided interfaces{Construct construction; Destruct destruction;}
35  reactors{ variables{Long density; Long T = 10; Long D = 30;}}
36 bindings{
37   bind(appearance.instantiate, construction);
38   bind(disappearance.destroy, destruction);
39 }}
```

Listing 1. Modularizing traffic congestion

To be able to utilize gummy modules, it is necessary to initialize their required interfaces, and the possible constraints among them. We provide a configuration language to express these; Listing 2 shows an example. Here, we assume that there are two road segments, for which the emergent behavior traffic congestion must be represented via instances of the gummy module `TrafficCongestion`. The variables `cong1` and `cong2` in line 3 are defined for this matter.

```
1 configurations{
2  initializations{
3   TrafficCongestion cong1, cong2;
4   cong1.appearance.publishers = [1]; cong1.disappearance.publishers = [1];
5   cong2.appearance.publishers = [2]; cong2.disappearance.publishers = [2];
6  }
7  constraints { precede (cong1, cong2); }
8 }
```

Listing 2. A configuration of gummy modules

Line 4 initializes the required interface of `appearance` of `cong1`, such that it filters the events of the type `SegmentEvent` published by the segment with the identifier `1`. The interface `event` of `appearance` is left unbound; therefore, runtime environment makes use of type matching to bind the interface to the events that are published to the environment.

Likewise, the required interface of other sub-gummy modules are initialized. The request to display or log traffic congestion data can be issued from Java objects by means of events. Whenever an event of the type `LogEvent` is published, the runtime environment bounds it to the required interface `event` of `cong1.utilization` and `cong2.utilization`. Since both gummy modules `cong1` and `cong2` select events of type `SegmentType` and `LogEvent`, line 7 defines the order in which these events must be processed by these modules; i.e. `cong1` must process them first.

5. Conclusion and Future Work

In this paper, we discussed the need for dedicated linguistic constructs to modularly represent behavioral patterns and their lifetime semantics in software systems. We explained that lack of such constructs obliges software engineers to adopt various design patterns and/or to provide workaround code, which complicates the implementations. We explains the concept of event-based modularization and our recent work termed as gummy modules, which facilitate representing a behavioral pattern as one holistic module that encapsulates its lifetime semantics. As future work, we would like to apply gummy modules to represent various kinds of behavioral patterns in different domains such as self-energy-adaptive systems. We would like to develop advanced composition mechanisms, such as inheritance and aggregation, for gummy modules.

References

[1] Compose*. http://composestar.sourceforge.net/.

[2] Esper. http://esper.codehaus.org/.

[3] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[4] K. Hoffman and P. Eugster. Cooperative Aspect-Oriented Programming. *Sci. Comput. Program.*, 74:333–354, March 2009.

[5] H. Hüni, R. Johnson, and R. Engel. A Framework for Network Protocol Software. In *OOPSLA '95*. ACM, 1995.

[6] T. Kamina, T. Aotani, and H. Masuhara. EventCJ: a Context-oriented Programming Language with Declarative Event-based Context Transition. In *AOSD '11*. ACM.

[7] S. Malakuti. Complex Event Processing with Event Modules. In *Reactivity, Events and Modularity workshop, co-located with SPLASH*. 2013.

[8] S. Malakuti and M. Aksit. Evolution of Composition Filters to Event Composition. In *SAC' 12*. ACM Press.

[9] S. Malakuti and M. Aksit. Event-based Modularization of Reactive Systems. In *Concurrent Objects and Beyond (to appear)*, LNCS. 2013.

[10] S. Malakuti and M. Aksit. Event Modules: Modularizing Domain-Specific Crosscutting RV Concerns. In *TAOSD (to appear)*, LNCS. 2013.

[11] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An Overview of the MOP Runtime Verification Framework. *International Journal on Software Techniques for Technology Transfer*, pages 249–289, 2011.

[12] M. Mezini and K. Ostermann. Conquering Aspects with Caesar. In *AOSD' 03*. ACM Press.

[13] H. Rajan and G. Leavens. Ptolemy: A Language with Quantified, Typed Events. In *ECOOP '08*, LNCS.

[14] W. Vanderperren, D. Suvée, M. A. Cibrán, and B. De Fraine. Stateful Aspects in JAsCo. In *Software Composition*. LNCS, 2005.