# Evolution of Composition Filters to Event Composition

Somayeh Malakuti and Mehmet Aksit
Software Engineering group, University of Twente, 7500 AE Enschede, the Netherlands
{s.malakuti,m.aksit}@ewi.utwente.nl

## ABSTRACT

Various different aspect-oriented (AO) languages are introduced in the literature, and naturally are evolved due to the research activities and the experiences gained in applying them to various domains. Achieving *modularity*, *composability* and *abstractness* in the implementation of crosscutting concerns are typical requirements that these languages aim to fulfill; and the degree to which they are fulfilled differs per language. Therefore, we always face two questions: what are the limitations of current AO languages from the perspective of these requirements, and what kinds of changes and/or new language mechanisms are necessary to address the limitations. This paper elaborates on the limitations of the current AO languages by means of runtime enforcement as an example domain. Via a new computation model termed as Event Composition Model, which is a successor of the Composition Filters Model, we outline the new language mechanisms that are necessary to overcome the limitations. This paper introduces the EventReactor language as an implementation of Event Composition Model, and by means of an example runtime enforcement technique, it illustrates the suitability of Event Composition Model to achieve better modularity, composability and abstractness in the implementation of concerns.

## Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Modules, packages*

## General Terms

Languages, Design

## Keywords

modularity, composability, abstractness, aspect-orientation, runtime enforcement

## 1. INTRODUCTION

Various different aspect-oriented (AO) languages are introduced in the literature [7, 1, 10, 3, 12, 2], and naturally are evolved due to the research activities and the experiences gained in applying them to various domains. Achieving *modularity*, *composability* and *abstractness* in the implementation of crosscutting concerns are typical requirements that AO languages aim to address. For this matter, these languages offer first-class abstractions to define aspects, so that the modularity is achieved; and offer various operators to compose aspects with the other concerns in program. Some AO languages [7] also aim at defining aspects at the higher level of abstraction without incorporating unnecessary implementation details. The degree to which the above-mentioned requirements are fulfilled differs per language. Therefore, we always face two questions: what are the limitations of current AO languages from the perspective of these requirements, and what kinds of changes and/or new language abstractions are necessary to address the limitations.

This paper makes use of runtime enforcement (RE) [4] as an example domain to answer these questions. RE techniques enable software to tolerate failures and to continue operating in case of failures. In these techniques, the changes in the states of the software are verified against the formally specified properties of the software. If any failure is detected, diagnosis and recovery actions may be performed to respectively detect the causes of the failure and to recover the software from the failure. Due to the difficulty of creating fault-free software, RE techniques are more and more adopted in large-scale software and this trend seems to continue also in the future. This causes the implementation of RE techniques also becomes extremely complex.

This paper identifies the core concerns that typically exist in RE techniques. By means of an example, the paper illustrates the need to fulfill the modularity, composability and abstractness requirements in the implementation of RE techniques, so that we can cope with the complexity of these techniques. This paper explains that the existing AO languages fall short to fulfill these requirements; nevertheless, the Composition Filters Model (CFM) [1] and its language Compose* [7] offer promising features for this matter. We introduce a new computation model termed as **Event Composition Model**, as a successor of the CFM, which offers a set of novel linguistic abstractions to overcome the identified shortcomings.

We introduce the EventReactor language, as a successor of Compose*, to implement Event Composition Model. By

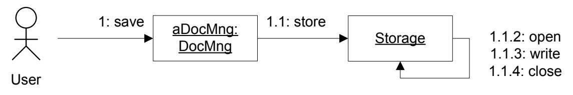

**Figure 1: The sequence of method invocations to store a document**

means of an example RE technique, we illustrate the suitability of Event Composition Model and its implementation language EventReactor to achieve modularity, composability and abstractness in the implementation of concerns.

This paper is not only useful for the composition filters practitioners but also for other AO language users, because the limitations of composition filters are also valid for most other AO languages. The evolution experience of the CFM may inspire the researchers to extend the current AO languages. The AO language designers can consider the challenges that exist in the implementation of RE techniques to introduce new language mechanisms if necessary.

The rest of this paper is organized as follows. Section 2 provides background information about RE techniques. Section 3 identifies the requirements to be fulfilled in the implementation of RE techniques and evaluates AO languages with respect to these requirements. Section 4 explains Event Composition Model, and Section 5 illustrates the features of the EventReactor language. Section 6 evaluates the suitability of the EventReactor language with respect to identified requirements, and Section 7 outlines conclusion and future work.

## 2. BACKGROUND

In the following, we first provide an RE technique that is used as the illustrative example throughout the paper, and then we identify the core concerns that typically exist in RE techniques.

### 2.1 An Illustrative RE Technique

Assume that there is a document-editing software, with two core modules *DocMng* and *Storage*, which provide services to edit a document and to save its contents, respectively. *DocMng* is implemented in Java, and *Storage* is implemented in C. Assume that it is required to verify at runtime that a request to save a document by the user eventually results in storing the document on the file system.

Figure 1 makes use of a UML collaboration diagram to depict the sequence of causally-dependent invocations that handles this request. To save a document, first the user invokes the method *save* on the object *aDocMng* of type *DocMng*. This causes the functions *store*, *open*, *write* and finally *close* to be invoked on *Storage*. For the sake of brevity, we eliminated the objects that facilitate inter-language communication.

We consider it as a failure if after the invocation of *save* by the user, (a) any of the other invocations does not occur in the specified order, and/or (b) the operating system thread in which a user's request is handled terminates before these events occur. As the recovery actions for the case (a), we would like to first log an error message, and then prevent the execution of the method whose invocation violates the specified sequence. For the case (b), we only want to log an error message.

### 2.2 Concerns in RE Techniques

We divide the concerns that typically exist in RE techniques in five categories: **base**, **verification**, **diagnosis**, **recovery** and **constraints**.

The **base** concerns are the concerns of interest in software, whose properties must be enforced. These are for example, objects, functions, subsystems, processes, or groups of them. In our example, *DocMng* and *Storage* are two base concerns. The changes in the states of the operating system thread must also be considered while verifying the behavior of *DocMng* and *Storage*. Therefore, the operating system thread is another base concern of interest in the software. These three form a group of correlated base concerns.

The **verification** concerns define the expected and/or unexpected properties of the base concerns, receive the necessary data from the base concerns, and verify the data against the specified properties. In our example, the specified sequence of invocations is a property to be enforced, and the functionality to check the specified sequence is provided by a verification concern. The verification of the specified properties results in new data, for example, indicating whether the properties are satisfied or violated. This data can be used by the diagnosis and recovery concerns.

The **diagnosis** concerns define the rules to diagnose causes of failures; for this matter, they may refer to the base concerns and/or the results of verification. The diagnosis also results in new data indicating the results of the diagnosis.

The **recovery** concerns define a set of actions to recover the base concerns from the failures, and for this matter they may refer to the data provided by the other concerns. In our example, the functionalities to log an error message and to prevent the execution of a method are provided by recovery concerns, which are executed if the verification concern reports the specified sequence of invocations is not satisfied. For the sake of brevity, we do not consider diagnosis concerns for our example RE technique.

The **constraint** concerns define the inter-dependencies within and/or among the other RE concerns. In our example RE technique, the order in which the two recovery concerns must be executed represents a constraint.

To be able to create a more adaptive system, it may be necessary to consider an RE technique as a base concern and to define higher-order RE techniques on the top of it. Such hierarchal organizations are quite common in adaptive control systems for example, where multi-levels of control systems can be stacked on each other. For this reason, any concern may be regarded as a base concern including for example the verification, diagnosis, and recovery concerns.

## 3. IMPLEMENTING RE TECHNIQUES

Due to the difficulty of creating fault-free software, RE techniques are more and more adopted in large-scale software and this trend seems to continue in the future. This causes the implementation of RE techniques also becomes extremely complex. In this section, we first identify the requirements that we claim must be fulfilled in the implementation of RE techniques, so that we can cope with the complexity of the RE techniques. Afterwards, we evaluate AO languages with respect to these requirements to identify their suitability for implementing RE techniques.

### 3.1 Requirements

We claim that a language is suitable for implementing an RE technique if it fulfills the following three requirements in the implementations:

- **Modularity of implementations**: Individual RE concerns must be represented as individual reusable modules, preferably by having a one-to-one correspondence between the elements of the language and the RE concerns. Otherwise, the implementation of a concern may be scattered across and tangled with the implementation of other concerns. Scattering and tangling are well-known problems that are discussed in the aspect-oriented literature. They decrease the modularity of the concerns, decrease the reusability of the concerns and increase the required effort to maintain software.

  For example for our illustrative RE technique, the employed language must enable us to represent *aDocMng*, *Storage* and the operating system thread as one group of correlated base concerns. Object-oriented languages provide a first-class abstraction to represent objects; whereas, the other kinds of concerns such as groups of correlated objects cannot directly be represented in these languages. As a result, we have to provide a workaround representation, which scatters across other concerns in the software.

- **Composability of implementations**: The composition mechanism offered by the language must offer a rich set of constructs for the following matters. a) To integrate various modularized concerns with each other. b) To express various kinds of composition constraints. c) To cope with different implementation languages of the concerns. d) To facilitate constructing higher-level of concerns by systematically reusing the existing ones, this facilitates considering RE concerns as base concerns and define a higher order RE on top of them.

  To implement our example RE technique, the composition mechanism of the employed language must enable us to compose the verification and recovery concerns with *DocMng* and *Storage* that are implemented in different languages. The language must also facilitate composing two recovery concerns with each other so that their order of execution can be constrained.

- **Abstractness of implementations**: The language must enable us to implement the concerns of interest naturally at the right level of abstraction without incorporating unnecessary implementation context (e.g. implementation language). This helps to increase the portability and reuse of concerns.

  For example, our illustrative software may evolve such that the module *Storage* is replaced with another module that provides the similar functionality, but is implemented in the Java language. We expect that the employed language enables us to reuse the implemented verification and recovery concerns, regardless of this change in the implementation language of *Storage*.

## 3.2 Shortcomings of Current AO Languages

The RE concerns are by nature crosscutting. For example, the verification concerns crosscut the base concerns to gather the necessary data from them and to verify their properties. The recovery concerns crosscut the verification concerns to get the result of the verification, and crosscut the base concerns to enforce their properties.

AO languages are introduced in the literature [10, 3, 12, 7, 2] to modularize crosscutting concerns as **aspects**. These languages facilitate the composition of aspects with the other concerns, and usually provide various constructs to constrain the compositions. It seems to us that the AO languages provide promising features to fulfill the modularity and composability requirements in the implementation of RE techniques. For example, a verification concern can be implemented as an aspect that is composed with the base concerns in software to check their properties; a recovery concern can be implemented as another aspect that is composed with the verification aspect and with the base concerns in software; and so on.

Several AO languages [10, 3, 12, 2] exist in the literature, which adopt the constructs of an existing programming language such as the Java, C and .Net languages. If these AO languages are employed to implement an RE technique, the abstractness requirement will not be fulfilled, because the implementation of concerns is specific to one programming language. Moreover, the shortcoming in fulfilling the abstractness requirement causes the modularity and composability requirements to be neglected too. Consider the following example.

We would like to verify the sequence of invocations shown in Figure 1. Since *aDocMng* and *Storage* are respectively implemented in Java and C languages, we must define two aspects, say one in AspectJ [10] and one in AspectC [3], to implement the verification concern. The former verifies the invocation of *save* on *aDocMng*, and the latter verifies the invocations of *store*, *open*, *write* and *close* on *Storage*. We also have to implement a program to compose these two aspects with each other, such that it is ensured that the sequence of invocations on *Storage* is causally dependent to the invocation of *save* on *aDocMng*. This solution however violates the composability requirement, because there is no standard linguistic mechanism to compose these two aspects that are implemented in two languages. Implementing a composition program for each different kind of composition is a costly and error-prone task, and may lead to the solutions that are not reusable.

To fulfill the abstractness requirement, one may consider employing a language-independent AO language. Compose* is an illustrative example of such languages. Although Compose* fulfills the abstractness requirement, it falls short in satisfying the other two requirements. The shortcomings are mainly rooted in the Composition Filters Model (CFM) that is underlying computation model of Compose*. In the following, we briefly explain the CFM and elaborate on shortcomings of the CFM and Compose*.

The CFM aims at improving the composability of object-oriented software. In such software, objects send messages between each other for example in the form of method calls. In the CFM, these messages can be **filtered**. Each filter has a **type**, which implements the functionality that should be executed if the filter receives a message. Filters are grouped in so-called **filter modules**. A **superimposition selector** chooses a set of classes using a query language and applies (superimposes) a specified filter module to them. As a result, all messages sent to and received by all instances of those selected classes are subjected to the filters within the filter module.

The CFM can be applied to any language that supports the notion of message passing between objects. In a non-object-oriented language such as C, the invocations of functions can be considered as messages that are passed between source files. This characteristic of the CFM helped the designers of the Compose* language to keep the language independent from any programming language.

Dedicated filter types can be provided to implement the concerns that exist in various domains. Multiple instances of a concern can be created via filters. For example for the purpose of RE, one may define a dedicated filter type that verifies messages against a specified sequence of messages, and reports the result of the verification. Dedicated filter types can also be provided to implement diagnosis and recovery concerns. This feature helps to preserve a one-to-one correspondence between the concerns of interest and the first-class abstractions of the Compose* language, so that the modularity requirement is fulfilled.

The composition offered by the CFM is limited to apply a filter module on an individual application object, and to process the messages that are sent or received by the application object. This degrades the degree to which the composability and consequently the modularity requirements are fulfilled, in the following two ways.

First, there are various different kinds of concerns in software, whose behavior must be verified and enforced; groups of correlated objects, processes, subsystems are examples. To employ Compose* for implementing an RE technique, we have to provide a workaround to represent these concerns as an application object. As it is explained earlier, this violates the modularity requirement in which a one-to-one correspondence between the concerns of interest and the first-class abstractions of the language is expected.

Second, if individual RE concerns are implemented as individual filters, we must be able to compose such filters with each other such that the overall RE technique is achieved. In the CFM, filters can be composed in one filter module to process messages in a sequence; however, we require more complex forms of composition for implementing an RE technique. For example, a recovery filter must be informed of the results of the verification performed by a verification filter and accordingly must take an action. Since such a composition of filters is not possible in the CFM, we have to sacrifice the modularity of the concerns and define them as one filter. Such a filter must also implement the desired composition constraints among concerns, if any.

## 4. EVENT COMPOSITION MODEL

As it is explained in the previous section, the CFM provides promising features for fulfilling the modularity and abstractness requirements. However, its composition mechanism must significantly be extended to satisfy the composability requirement too. This paper introduces a computation model termed as **Event Composition Model**, as a successor of the CFM to overcome its shortcomings.

The interactions among the concerns of RE techniques have by nature a transient characteristic, which means *changes in the states* of a concern drives the other concerns. Definition of a state change can be different. For example, it can refer to an invocation of a method on an object, calling a function, begin or end of a thread of execution, a success or failure of a verification process, triggering a diagnosis process, committing a recovery action, etc. For example, the verification concerns observe the changes that occur in the states of base concerns, and verify them against the specified properties of the base concerns. If the verification of a property fails, it may trigger a diagnosis and/or a recovery concern.

In Event Composition Model, all such state changes are termed as **event**. Although the notion of event seems to be a fundamental concept for RE techniques, it is a too low-level representation with respect to the concerns of interest. For example, it may be necessary to represent all the events that are related to an object, a function, a thread of execution, a process, or a subsystem as a linguistic abstraction. It is therefore logical to consider a group of *related* events as a *module*, which we term as **event module**.

In the literature [6], a module is defined as a software unit with input and output interfaces. The former defines the services that the module requires from its context; the latter specifies the services that the module provides for its context. A module promotes information hiding by separating its interface from its implementations.

We think that like the modules in programming languages, an event module must be uniquely identifiable for example by its name, must provide input and output interfaces and must separately specify its implementations.

The input interface of an event module is defined by the events that it groups. The input interface is invoked *implicitly*, which means upon the occurrence of a grouped event, the corresponding implementations of the event module are invoked without explicitly writing a code for it.

The implementations of event modules are termed as **reactors**. Reactors are grouped in a module termed as **reactor chain**. Such reactors are composed with each other such that they process the events in a sequence starting from the first specified reactor within the reactor chain until the last reactor. One or more reactor chains can be bound to an event module as its implementations.

Each reactor has a **type** implementing the operation that should be executed if the reactor receives an event. Each reactor type may publish new events during its operation. These are termed as **reactor events**. The output interface of an event module is a union of the reactor events that are published by the reactors bound to the event module.

The selection of events, and the grouping of the events in an event module is carried out by an **event composition language**. The language is capable of selecting any event that is declared in the system and is in the scope. The reactor events can also be selected, and can be specified as the input interface of other event modules. This enables us to create more abstract event modules by systematically composing the existing ones. The compositions may be constrained; the constrains is defined using an **event constraint language**.

If Event Composition Model is employed to implement an RE technique, a concern of interest can be represented as an event module. In this case, the set of events that the concern requires from other concerns are specified as the input interface of an event module. The set of events that the concern provides to the other concerns are specified as the output interface of the event module. The implementation of the concern is provided via reactors.

Reactors, reactor types and reactor chains resemble filters, filter types and filter modules in the CFM, respectively. In contrary to Event Composition Model, the CFM offers a limited sort of event processing. In the CFM, only two kinds of events are supported; these are the events corresponding to the incoming and outgoing messages that are exchanged among application objects. However, Event Composition Model is open-ended with respect to the kinds of supported events. The composition mechanism of the CFM is limited to superimpose individual instances of filters (modules) on individual objects. However, the event composition language in Event Composition Model facilitates grouping events that are published by single and/or multiple correlated publishers. In Event Composition Model, reactor types can publish events; this cannot be realized by filter types in the CFM. Finally, superimpositions in the CFM are not named; therefore, it is not possible to refer to them. In Event Composition Model, however, event modules are named, and the events in their output interface can be selected by the event composition language.

## 5. EVENTREACTOR: A LANGUAGE FOR EVENT COMPOSITION MODEL

There are several languages that support the notion of events, but as we discuss in details in [11], these languages fall short to support Event Composition Model. Therefore, we introduce the EventReactor language, which is a successor of Compose*, as the implementation of Event Composition Model. The language supports predefined kinds of events and provides an API to programmers to declare new kinds of events. EventReactor provides dedicated linguistic constructs to define event modules, reactor types, reactors and reactor chains. It makes use of the Prolog language and the set operators [5] as its event composition language. Dedicated constructs are provided by the language to define composition constraints.

Adopted from Compose*, the EventReactor language does not make any assumption about the implementation language of software, and its compiler can support software implemented in the Java, C and .Net languages. The syntax of the language, its compiler and execution semantics are explained in details in [11]. Due to the space limit, this paper only explains the features of the EventReactor language by providing an implementation of our illustrative RE technique.

For implementing the illustrative RE technique, the following tasks must be carried out: a) the events of interest must be defined in the language, and must be published to the runtime environment of the EventReactor language, b) dedicated reactor types must be provided to implement the functionality of the RE concerns, and c) event modules and reactor chains must be defined to implement different concerns that exist in the RE technique. In the following, these tasks are explained in detail.

### 5.1 Defining and Publishing the Events

In EventReactor, each kind of event is represented via a set of attributes that are categorized in two groups: **static context** and **dynamic context**. The attributes whose values are known when a new kind of event is defined, are in the category of static context. The attributes whose values are known when an event is published, are in the category of dynamic context. Each kind of event must at least define two attributes named *uid* and *PrologFacts* as its static context. The former is internally used by the EventReactor language to uniquely identify the event kind in the language, and the latter is a set of Prolog facts that are used to define an event kind in the language and later on to select events from the language.

The EventReactor language supports predefined kinds of events, which correspond to the following state changes: a) before invocation of methods, b) after invocation of methods, c) after invocation and immediately before execution of methods, and d) after execution of methods, which have terminated normally. These are considered as predefined, because the compiler of the EventReactor language identifies them in program, and defines them in the language.

Listing 1 shows an example code that compiler uses to declare a predefined event kind in the EventReactor language. Line 1 defines the variable `ekind` of the type `EREvent` that is a class provided by the EventReactor language. In line 2, the compiler specifies the value `'e1'` as the unique identifier of the event kind in the language. Lines 3 to 9 define the attribute `PrologFacts`.

```
1  EREvent ekind = new EREvent();
2  ekind.staticcontext.add("uid", " 'e1' ");
3  ekind.staticcontext.add("PrologFacts",
4   "isBeforeExecution('e1','public void save(java.lang.Object)').
5   isMethodWithName ('public void save(java.lang.Object)', 'save').
6   isClassWithName ('public class DocMng extends java.lang.Object',
7          'DocMng').
8   isDefinedIn ('public void save(java.lang.Object)',
9          'public class DocMng extends java.lang.Object').");
10 ekind.dynamiccontext.add("threadid", "");
11 ekind.dynamiccontext.add("method", "");
12 EventReactor.declare(ekind);
```

**Listing 1: Declaring a predefined event**

The expression `isBeforeExecution` in line 4 specifies that the event kind represent the events that correspond to the state change after the invocation of a method and immediately before the execution of the method. The first argument is the unique identifier of the event kind, and the second argument is the signature of the method of interest. The character `'.'` in Prolog represents the termination of a fact.

Compose* provides various Prolog expressions, which are also adopted by EventReactor, to identify the methods and the classes of interest in program. For the sake of brevity, we show a subset of these Prolog facts in Listing 1. The expression `isMethodWithName` in line 5 specifies the method of interest. The first argument is the signature of the method, and the second argument is the name of the method. The expression `isClassWithName` in line 6 specifies the class `DocMng`. The first argument is the signature of the class, and the second argument is the name of the class. The expression `isDefinedIn` in line 8 specifies that the method `save` is defined in the class `DocMng`.

When defining a new kind of event, we must also define the list of attributes that represent the dynamic context of the event kind. The compiler considers two attributes *threadid* and *method* as the dynamic context of the predefined event kinds. The attribute *threadid* will keep the unique identifier of the thread of execution in which a predefined event occurs. The attribute *method* will keep the reflective information of the method that a predefined event corresponds to. These attributes are defined in lines 10 and 11. Line 12 defines the event kind in the language. The other predefined event kinds are declared in the language in a similar way.

The EventReactor language also provides an API to programmers to define new kinds of events. For our example, we make use of the code excerpt in Listing 2, to define an event kind for the events representing the termination of a thread of execution. Line 1 defines the variable `ekind`, and line 2 specifies `'e2'` as the unique identifier of the event kind. The compiler of the EventReactor language reports an error if the assigned identifier is not unique in the language. Lines 3 and 4 define the Prolog fact `isEventWithName('e2', 'terminated').`, which specifies `'terminated'` as the name of the event kind. Line 5 defines the attribute *id* that will keep the unique identifier of the thread whose execution is terminated. Line 6 defines the event kind in the language.

```
1 EREvent ekind = new EREvent();
2 ekind.staticcontext.add("uid", " 'e2' ");
3 ekind.staticcontext.add("PrologFacts",
4                     "isEventWithName('e2', 'terminated').");
5 ekind.dynamiccontext.add("id", "");
6 EventReactor.declare(ekind);
```

**Listing 2: Declaring a user-defined event**

For the predefined events, the compiler modifies the program code to assign the expected values to the attributes *threadid* and *method*, and to publish the events to the runtime environment of the EventReactor language. For user-defined events, this must be carried out by the programmers. Listing 3 shows an excerpt of the code that publishes the event defined in Listing 2. Line 2 of Listing 3 specifies `'e2'` as the unique identifier of the event. This enables EventReactor to match a published event with a declared event in the language. Line 3 specifies the value of the attribute *id*, and line 4 publishes the event. This code must be inserted in places in the program where the termination of a thread of execution is detected.

```
1 RTEvent event = new RTEvent();
2 event.staticcontext.add("uid", " 'e2' ");
3 event.dynamiccontext.add("id", getTerminatedThreadID());
4 EventReactor.publish(event);
```

**Listing 3: Publishing a user-defined event**

It is note-mentioning that the runtime environment of EventReactor is implemented in Java, and the API to publish events is available in Java, .Net and C languages. This API makes use of Java-JNI technique [9] to announce the events to the runtime environment.

## 5.2 Defining the Reactor Types

To implement our illustrative RE technique in the EventReactor language, we provide four reactor types *React*, *RegularExpression*, *Log* and *ForceReturn*. Reactor types are defined in the EventReactor language in a similar way as filter types are defined in Compose*. The implementation details can be found in [7].

The only function of the reactor type *React* is to publish a reactor event when it receives an event to process. The name of the reactor event may be provided as an argument to the reactor type; otherwise, it has the same name as the event being processed. The reactor type *RegularExpression* receives a regular expression as its parameter, and translates it to a deterministic finite state automaton according to the algorithm discussed in [8]. It makes use of the automaton to check an event against the regular expression formula, and publishes the reactor event *violated* if the event does not satisfy the formula. The reactor type *Log* reports a message on the screen when it receives an event to process. The message is passed to the reactor type as an argument. The reactor type *ForceReturn* prevents the execution of a method by returning the flow of execution to the caller of the method. The information about the method is provided as the dynamic context of a predefined event.

## 5.3 Implementing the RE Technique

Individual concerns of our illustrative RE technique can be implemented as individual event modules, which are composed with each other. We start from the base concerns *DocMng*, *Storage* and the thread of execution in which the specified events occurs. Listing 4 represents these correlated concerns as one event module. Starting from line 1, EventReactor provides the construct `eventpackage` as a means to package a set of event modules. In this example, the event package is named `base_concern`. The events of interest are specified in the part `selectors` of the event package. Line 3 selects the events `E` whose name matches the string `'terminated'`, and names them as `e_terminated` in the event package. This Prolog expression is defined in the language via Listing 2.

In line 4 the Prolog expression `isBeforeExecution (E, M)` selects the predefined events `E`, which correspond to the state change after the invocation and immediately before the execution of the methods `M`. The Prolog expressions in lines 5 to 7 select the methods `M` whose name matches the string `'save'`, and are defined in the classes whose name matches the string `'DocMng'`. The character `','` between the Prolog expressions is a conjunction operator. The results of these Prolog queries are named as `e_save` in the event package. Similarly, lines 8 to 11 select the other predefined events that occur on `Storage`. It is worth mentioning that EventReactor supports wildcard characters in the Prolog expression to enable us to select various numbers and/or kinds of events.

```
1  eventpackage base_concern{
2   selectors
3    e_terminated = {E | isEventWithName(E, 'terminated')};
4    e_save = {E | isBeforeExecution(E, M),
5            isMethodWithName(M,'save'),
6            isClassWithName(C,'DocMng'),
7            isDefinedIn(M, C)};
8    e_store = ...
9    e_open = ...
10   e_write = ...
11   e_close = ...
12  eventmodules
13   base := {e_save, e_store, e_open, e_write, e_close, e_termination}
14           <- perthread {group};
15 }
```

**Listing 4: An event module for base concerns**

Event modules are defined in the part `eventmodules` of an event package. EventReactor makes use of the set operators to group the selected events as the input interface of the event modules, and provides the operator <- to bind reactor chains to event modules. EventReactor supports various instantiation strategies for event modules, which are explained throughout the paper.

Lines 13 and 14 of Listing 4 define the event module base, which specifies perthread as its instantiation strategy and the reactor chain group as its implementation. We assume that a request to save a document must be handled in one thread of execution. Multiple requests may be handled by multiple threads. The keyword perthread indicates that individual instances of the event module must be created for each individual thread of execution in which the selected predefined events occur. The details of filtering events based on the instantiation strategy of event modules can be found in [11].

Listing 5 shows the reactor chain group, which defines the reactor forward of type React. At runtime when any of the specified events in Listing 4 occurs, the input interface of the event module base is activated, and the event is provided to the reactor forward. Consequently, a reactor event which has the same name as the event is published.

```
reactorchain group{
 reactors
   forward: React;
}
```

**Listing 5: A reactor chain for base concerns**

As the next step, we would like to implement a verification concern, which checks whether a request to save a document is handled correctly. Listing 6 defines the corresponding event module. Lines 3 to 10 select all the events that form the output interface of the event module base. These are in fact the reactor events that are published by the reactor forward. Lines 12 to 15 define the event module verification. Here, all selected events are specified as the input interface, perinstance is specified as the instantiation strategy, and verify is specified as the implementation of the event module. The expected sequence of events to handle the user's request is specified as a regular expression formula and is passed as an argument to the reactor chain verify. The regular expression indicates that the event eb_save must be followed by eb_store, eb_open, one or more times eb_write and finally eb_close, and this sequence may occur zero or more times. Since the instantiation strategy perinstance is chosen, separate instances of the event module will be created for separate instances of the event module base that publish the selected reactor events.

```
1  eventpackage verification_concern{
2   selectors
3     eb_save = {E | isEventWithName(E, 'e_save'),
4                  isEventModuleWithName(EM, 'base_concern.base'),
5                  isPublishedBy(E, EM)};
6     eb_store = ...
7     eb_open = ...
8     eb_write = ...
9     eb_close = ...
10    eb_terminated = ...
11  eventmodules
12    verification :=
13     {eb_save,eb_store,eb_open,eb_write,eb_close,eb_terminated} <−
14       perinstance
15     {verify('(eb_save eb_store eb_open eb_write+ eb_close)∗')};
16 }
```

**Listing 6: An event module for verification concern**

Listing 7 shows the reactor chain verify, which receives a parameter named ?regformula, and defines the reactor regexp from type RegularExpression. In the body of the reactor, the reactor parameter formula is assigned with ?regformula. If any of the expected event does not occur in the specified order in the regular expression, the reactor regexp publishes the event violated.

Listing 8 defines the event modules log and prevention, which implement the recovery concerns of our example. The event violated, which is published by the event module verification, is selected and is specified as the input interface of these event modules. The reactor chains log_recovery and prevent_recovery are respectively bound to the event modules log and prevention. The event modules are specified to be instantiated as singleton, because the recovery actions are stateless. In the part constraints of the event package, the composition constraints are specified for the event modules. The keyword precede specifies that the event module log must process the event e_verification before the event module prevention. Listing 9 defines the reactor chains log_recovery and prevent_recovery.

```
reactorchain verify(?regformula){
 reactors
   regexp: RegularExpression { reactor.formula = ?regformula; };
}
```

**Listing 7: A reactor chain for verification concern**

```
eventpackage recovery_concern{
 selectors
   e_verification= {E | isEventWithName(E, 'violated'),
               isEventModuleWithName(EM, '∗.verification'),
               isPublishedBy(E, EM)};
 eventmodules
   log := {e_verification} <− singleton {log_recovery};
   prevention := {e_verification} <− singleton {prevent_recovery};
 constraints
   precede(log, prevention);
}
```

**Listing 8: Event modules for recovery concern**

```
reactorchain log_recovery{
 reactors
   logger: Log {reactor.info = 'An error has occurred!'; }
}
reactorchain prevent_recovery{
 reactors
    preventer: ForceReturn;
}
```

**Listing 9: Reactor chains for recovery concern**

Assume that the following scenario occurs at runtime. The method save is invoked on the object aDocMng in the thread of execution t. The execution of the method save is suspended, and the flow of execution is transfered to the runtime environment of EventReactor. The runtime environment identifies that the event e_save specified in lines 3 to 5 of Listing 4 has occurred. Since there is no instance of the event module base for the thread t, the runtime environment creates one, and forwards the event e_save to the reactor chain group and consequently to the reactor forward. The reactor publishes the reactor event e_save, which activates the input interface of the event module verification as it is specified in Listing 6. The runtime environment creates an instance of this event module, and forwards the event to the reactor `regexp`. The event is checked against the specified regular expression, which does not violate it.

Assume that in the same thread of execution, the function `open` is invoked on `Storage`. This causes the event `e_open` specified in Listing 4 to be detected and be processed in a similar way using the same instances of the event modules `base` and `verification`. Since, it violates the specified regular expression, the reactor event `violated` is published by the reactor `regexp`. This event activates the input interface of the event modules `log` and `prevention` as it is specified in Listing 8. The runtime environment creates a single instance of these event modules, and provides the event to the reactor `logger` that prints the specified error message on the screen. Afterwards, the event is provided to the reactor `preventer`.

In the EventReactor language, each reactor event keeps a reference to the original event; therefore, there will be a chain of events that are causally published after each other. Each predefined event also keeps a reference to its corresponding method/function in the program. In our example, the chain of events contains two events `e_open` and `violated`, which `e_open` keeps a reference to the function `open` in the program. The reactor `preventer` iterates through this chain, obtains the necessary reflective information about the function `open`, and informs the runtime environment that the execution of this function must be prevented. When the flow of execution returns to the function `open`, the runtime environment prevents its execution by returning the flow of execution to the caller of the function.

Assume that the thread of execution `t` terminates before all the specified events occur. This causes the event `terminated` to be published, which violates the specified regular expression. The recovery actions are executed, and because the event `terminated` is not a predefined event, `preventer` ignores it.

## 6. EVALUATION

As the listings in the previous section show, the abstractions offered by Event Composition Model, which are implemented by the EventReactor language, enable us to implement various different concerns that exist in an RE technique. The abstractness requirement is fulfilled in the implementations, because the EventReactor language does not make any assumption about the implementation language of program. The modularity requirement is fulfilled by means of event modules. The separation of reactor chains from the event modules increases the modularity of the implementations further.

The composability requirement is fulfilled too, because reactors can publish events, and these events can be selected as the input interface of other event modules. As a result, a hierarchy of event modules is formed in which the event modules at higher levels of the hierarchy abstract from the events modules locating at their lower levels. This increases the modularity, composability and the abstractness of implementations further. For example, an event module residing at higher levels of the hierarchy can remain unchanged, if its lower level event modules change, providing that the output interface of the lower level event modules remains the same.

As for Compose*, we believe that the other AO languages can also be extended to support this computation model, and consequently offer better modularization, composition and abstraction mechanisms.

## 7. CONCLUSION AND FUTURE WORK

This paper discussed that current AO languages usually aim at implementing crosscutting concerns such that the *modularity*, *composability* and *abstractness* requirements are fulfilled in the implementations. However, these languages fall short in satisfying these requirements; hence new language mechanism are required. This paper introduced a new computation model named as Event Composition Model and its implementation language EventReactor. By means of an example, the paper illustrated the suitability of the new computation model and language in fulfilling the identified requirements.

As future work, we consider utilizing the EventReactor language in various different domains, whose concerns have the same event-driven characteristics as the runtime enforcement concerns. We also consider extending EventReactor with various compile-time checks, for example, to ensure that multiple recovery concerns do not conflict with each other.

## 8. REFERENCES

[1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting Object Interactions using Composition Filters. In *ECOOP*, volume 791 of *LNCS*, pages 152–184. Springer-Verlag, 1993.

[2] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. Overview of CaesarJ. In *TAOSD*, LNCS, pages 135–173. 2006.

[3] AspectC. http://www.cs.ubc.ca/labs/spl/projects/aspectc.html.

[4] H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. J. Pace, G. Rosu, O. Sokolsky, and N. Tillmann, editors. *Proceedings of First International Conference on Runtime Verification*, volume 6418 of *LNCS*. Springer-Verlag, 2010.

[5] M. Ben-Ari. *Mathematical Logic for Computer Science*. Springer, 2003.

[6] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, and R. Little. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2002.

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

[8] J. E. Hopcroft, R. Motwani, and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2000.

[9] Java-JNI. http://download.oracle.com/javase/1.5.0/docs/guide/jni/.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *ECOOP*, pages 327–353, London, UK, UK, 2001. Springer-Verlag.

[11] S. Malakuti. *Event Composition Model: Achieving Naturalness in Runtime Enforcement*. PhD thesis, University of Twente, 2011.

[12] H. Rajan and K. Sullivan. Eos: Instance-Level Aspects for Integrated System Design. In *ESEC/FSE*, pages 297–306, Helsinki, Finland, 2003. ACM.