Complex Event Processing with Event Modules

Somayeh Malakuti[*]
Technical University of Dresden– Germany
somayeh.malakuti@tu-dresden.de

ABSTRACT

To effectively cope with the complexity of event processing application, there is a need for dedicated modularization and composition mechanisms for such applications. To this aim, we define a set of requirements that must be fulfilled by a language, and identify the shortcomings of a representative set of current languages with respect to these requirements. This paper discusses that event modules provide an inherent support to achieve modularity and compose-ability in the implementation of event processing applications. We explain a new implementation of event modules, and illustrate its suitability to fulfill the identified requirements.

Categories and Subject Descriptors

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

General Terms

Languages, Design

Keywords

event-based modularization, event-based composition, loose-coupling, separation of concerns

1. INTRODUCTION

There are various kinds of applications that deal with certain kind of event processing. Runtime verification techniques [10], self-adaptive software systems [13] and traffic monitoring systems are examples. In these applications, so-called *event processing agents* are defined, which mediate between *event producers* and *event consumers*, to perform various operations on the event streams.

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

Due to the multiplicity of event processing agents, event producers and consumers, the implementation of event processing applications can become complex. To effectively cope with the complexity, we claim that the separation of concerns and loose coupling among the concerns must be fulfilled in the implementations. Advanced programming languages offer various module abstractions and composition operators for these matters. To be able to evaluate the suitability of these abstractions, we define a set of modularity and composition requirements for event processing applications, and evaluate a representative set of languages with respect to these requirements.

In [8, 9], we introduced **event modules** as novel linguistic abstractions to modularly represent a group of correlated events and the reactions to them. We introduced the EventReactor language, which implements event modules. In this paper, we explain that event modules can effectively modularize the concerns that appear in event processing applications, and facilitate loose coupling in the compositions. We introduce a new implementation of event modules, which improves the compose-ability of event modules at instance level, facilitates both implicit and explicit composition of event modules, and improves the modularity of implementations by modularizing the event selection code.

This paper is organized as follows. Section 2 provides background information about event processing applications; Section 3 outlines a set of requirements for implementing such applications; Section 4 evaluates a representative set of languages; Section 5 discusses event modules and illustrates their suitability in achieving modularity and loose coupling in the implementation of event processing applications; and Section 6 outlines the conclusions and future work.

2. BACKGROUND

2.1 Event Processing Applications

Nowadays, there are numerous applications that deal with certain kind of event processing. Runtime verification techniques [10], self-adaptive software systems [13], traffic monitoring software systems are examples. Runtime verification techniques check the events that occur in software against the formally specified properties of the software, and detect the failures. Self-adaptive software systems monitor environmental changes, analyze them, and adapt themselves accordingly. Traffic monitoring software systems receive traffic flow information from the sensors that are embedded in roads, and reason about traffic flow in the roads.

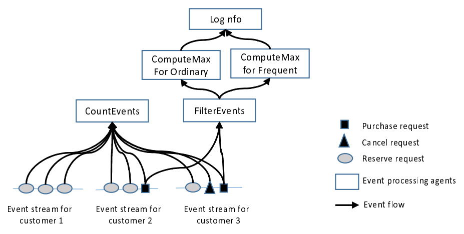

Figure 1: An example event processing application

At a high level of abstraction, there are four kinds of entities that play a role in implementing event processing applications [4]: *events*, *event producers*, *event consumers* and *event processing agents*. An event is usually defined as something that happens or is regarded as happening. As the name implies, event producers and event consumers are the entities that provide and receive events, respectively. Various kinds of software and hardware entities can be regarded as event producers and consumers. Event processing agents are software entities that mediate between event producers and event consumers to process the events. Event processing agents may have a crosscutting nature [7]; they may collect multiple events from one or more event producers and/or other event processing agents.

Three kinds of event processing are distinguished [4]: a) simple event processing, b) event stream processing, and c) complex event processing, which respectively looks at a single event, across multiple events, and across multiple event streams that may come from various producers. The processing may be *stateful* or *stateless*. In the former case, the result of event processing is influenced by the history of events; in the latter case, the result is only influenced by the event being processed.

2.2 Illustrative Example

We make use of a simplified online shopping application as an illustrative example. In this example, *customers* are actual event producers who issue events to purchase, reserve or return products; *salesmen* are the target event consumers who receive and handle the requests. As Figure 1 shows, there is a separate event stream for each costumer; the number of customers and consequently the number of event streams may change during the operation of the software.

As depicted in the figure, we would like to define a set of event processing agents to perform the following statistical operations on the event streams. Here, the event processing agent *CountEvents* records the number of purchase, reserve and cancel requests in each 20 minutes. *FilterEvents* classifies the purchase requests into the *ordinary* and *frequent* ones; if a customer issues the request to purchase a product more than *30* times in *10* days, it is considered a frequent request; otherwise, it is an ordinary one. The agents *ComputeMax* identify the maximum amount of purchase in the last *20* minutes for each category of frequent and ordinary requests. Finally, *LogInfo* records the information about the maximum amount of ordinary and frequent purchases.

3. REQUIREMENTS

To be able to effectively cope with the complexity and to increase the reusability and evolvability of event processing applications, we claim that two issues must be taken into account in the development of such applications: *separation of concerns* [1] and *loose coupling among the concerns* [4].

Separation of concerns implies that the concerns that are involved in a program must be identified and separated. In event processing applications, various kinds of computations may be performed on event streams, which may be regarded as the concerns of interest, and must be separated from each other and from the main application logic. Loose coupling implies that there is no need for event producers and consumers to be aware of each other; event producers must not depend on the computations performed by event processing agents and/or event consumers, and vice versa.

To be able to effectively separate the concerns that appear in event processing applications, and to compose them with each other such that they are loosely coupled, we claim that the abstractions offered by a programming language must fulfill the following requirements:

1. **Event representation**: Events are the core abstractions in event processing applications, which may be provided by different kinds of producers. This means that a language must provide suitable means to (a) define the events of interest, (b) detect their occurrence, (c) select them from event streams, and (d) provide them to event processing agents and event consumers. If a language falls short in these matters, programmers may be obliged to provide workaround code in the implementations, which may increase the complexity of the programs.

2. **Event-based modularization**: In the literature [2], modules are defined as reusable software units with well-defined required and provided interfaces, which encapsulate their implementation. In event processing applications, event producers, event processing agents and event consumers communicate with each other via events. To achieve a better separation of concerns, we claim that the module abstractions of a language must respect this characteristic of event processing applications, and facilitate the *event-based modularization* of these concerns by fulfilling the following requirements:

    (a) **Referrable identity**: To be able to refer to and reuse modules, a language must facilitate defining unique names for the modules and referring to the modules via their unique names.

    (b) **Event-based required interface**: It must be facilitated to define the required interface of modules in terms of the events that must be recieved from event streams and/or from other modules.

    (c) **Event-based provided interface**: Language must facilitate defining the provided interface of modules in terms of the events that are produced by them.

    (d) **Event processing function**: The language must offer suitable constructs to program various stateless/stateful simple event processing, event stream processing and complex event processing logics.

    If a language falls short in supporting event-based modularization of concerns, programmers may be obliged to provide workarounds using the available abstractions in the language; this may increase the complexity of implementations and reduce their modularity.

3. **Event-based composition**: To achieve a better separation of concerns, modules must be composed with each other at the interface level. Event-based modularization of concerns implies that a language must also offer suitable means to support event-based composition of the modules. We claim that a language must fulfill the following requirements:

   (a) **Loose coupling to event types**: During the lifetime of an application, the kinds of the events that must be processed by event processing agents and event consumers may change. To increase the reusability and evolvability of implementations, module interfaces must not be tightly coupled to the type of the events that they require or provide. Tight coupling to event types may lead to the redefinition of modules if the event types evolve.

   (b) **Loose coupling to event producers, event consumers and event processing agents**: During the lifetime of an application, the number of event producers, event processing agents and event consumers may change. This implies that a language must facilitate the composition of these at the interface level, such that the compositions can flexibly cope with the absence or presence of modules.

4. SHORTCOMINGS IN MODULARIZING EVENT PROCESSING APPLICATIONS

Advanced programming languages offer various kinds of module abstractions and composition operators to facilitate separation and composition of concerns in implementations. In this section, we briefly evaluate a representative set of languages with respect to the requirements defined in Section 3.

4.1 Object-Oriented (OO) Languages

OO languages offer *objects* as a means to achieve separation of concerns in the programs. Objects can communicate with each other via explicit method invocation and/or an event-based mechanism if possible in the adopted language.

4.1.1 Communication via Explicit Invocation

Let us focus on a possible implementation of event processing applications in which individual event producers, event processing agents and event consumers are defined as individual objects. We may consider adopting various design patterns [5], such as the Observer pattern, to achieve loose coupling among the objects.

In an Observer-based implementation of event processing applications, event producers get the role of *subject*, and event processing agents and/or event consumers get the role of *observer*. Multiple observer objects may register for a subject object, and are informed of the changes in the subject via an invocation to their so-called *notify* method. Such an invocation can be regarded as the occurrence of an event, and the arguments of the invocation can be regarded as the attributes of the event.

With respect to the requirement *event representation*, in a standard implementation of the Observer pattern, the type of events that can be produced is fixed in the definition of observer objects. Firstly, this creates a tight coupling between event producers and event consumers, because the producers are limited to provide the events specified by the consumers. Secondly, to support new types of events, new implementations of the Observer pattern must be introduced, which obviously complicates the implementations if large number of event types must be supported.

With respect to the requirement *event-based modularization of concerns*, the required interface of an observer object is fixed to support one kind of event. This makes the Observer-based implementation suitable for simple event processing, in which an observer object implements the functionality to process a single event. To implement stateless/stateful complex event processing, one has to define extra code to gather individual events from individual observer objects and to reason about the correlations among the events. Such an implementation of complex event processing functionality, however, scatters across multiple observer objects and/or methods, which may lead to complex implementations.

To enable an observer object to produce events as its provided interface, a new implementation of the Observer pattern must be provided in which the observer object gets the role of *subject*. Depending on the number of events that must be provided and the target recipient of the events, multiple implementations of the Observer pattern may be needed, which increases the complexity of the implementations further.

With respect to the requirement *event-based composition of concerns*, although observers and subjects are defined separately, there is an explicit tight coupling among them; an observer can only process event streams that are produced by the subjects to which it is bound. Consequently, if the application requirements change such that the number of subjects or observers changes, the binding must be redefined. Changing the number of observers may also cause redefinition of observers. For example, if two observers are bound to a subject, it may be necessary to define the order in which they must process events. To this aim, one may adopt the Mediator pattern, which requires redefining the observers to implement the Mediator pattern. Last but not least, if an observer object must collect events from multiple streams, multiple subjects must be defined as event producers. Consequently, the invocation to the observer object scatters across multiple subject objects.

4.1.2 Communication via Events

Advanced OO languages usually offer an event-delegate mechanism to facilitate implementation of event-based applications. This sections evaluates the event-delegate mechanism of C#; nevertheless the discussions can be generalized for the other event-delegate mechanisms that have similar characteristics.

In the event-delegate mechanism of C#, new event types and their attributes can be defined via special kinds of classes, which extend class *System.EventArgs*. To bind producers to consumers, C# provides a pointer-like mechanism named as *delegate*, which is a type that references a method; any method that matches the signature of a delegate can be assigned to the delegate. This mechanism facilitates binding various event consumers to an event producer. Events are published by instantiating the corresponding event type, and invoking the corresponding delegate.

Implementing event-based applications using such event-delegate mechanisms comes with certain shortcomings. With respect to the requirement *event representation*, although new kinds of event types can be programmed, they are limited to the ones that are published from within C# programs. However, as we studied in [8, 9] for the domain of runtime verification, the kinds of events that appear in event processing applications cannot be limited to one programming language. Supporting language-specific events obliges programmers to provide workarounds to map the desired events to the ones supported in C#, which may complicate the code and reduce modularity of implementations [8, 9].

With respect to the requirement *event-based modularization of concerns*, an event producer can be modularized via a class that defines events along with necessary delegates, and publishes events. An event consumer or an event processing agent can be implemented as a class that defines a method whose signature matches the desired delegate; this method implements the functionality to process the event. This means that the event-delegate mechanism of C# provides a natural support for implementing simple event processing applications, in which each event consumer method handles a single event. Implementing stateless/stateful complex event processing functionality leads to the same problems explained for the Observer-based implementation. With respect to the requirement *event-based composition of concerns*, as for the Observer-based implementation, there is an explicit tight binding between event consumers and event producers; this leads to the same problem explained in the previous subsection.

4.2 Aspect-Oriented (AO) Languages

Due to the crosscutting nature of event processing agents, one may consider adopting AO languages for their modularization. In an AO implementation of event processing applications, join points can be regarded as events; base objects in which join points are activated can be regarded as event producers. Aspects can be regarded as means to modularly represent event consumers and/or event processing agents. Here, the pointcut designators are means to select the events of interest; hence, they define the required interface of aspect module. Advice provides the functionality to process the events. The provided interface of an aspect module is formed around the set of join points that are designatable in the aspect module.

With respect to the requirement *event representation*, the set of supported events is defined by the join point model of the adopted AO language. Some AO languages such as AspectJ and Compose* support a fixed join point model. If desired events are not defined in the join point model, workaround mappings must be provided; this may increase the complexity and decrease the modularity of implementations. There are various proposals to support programmable join point models [6, 14], which are mainly limited to Java as the base language. As we studied in [8, 9], supporting a single base language may also reduce the modularity of implementations when events are published from various sources, for example multi-language base software.

With respect to the requirement *event-based modularization of concerns*, pointcut designators provide an inherent support to select the events of interest. The possibility to select the events of interest, which may occur in various streams, is influenced by the expression power of the pointcut designators and the instantiation strategy of the adopted language. If the expression power of pointcut designators is limited, one has to provide workaround code in advice code to express desired event selection semantics. Consequently, the specification of module interfaces gets tangled with the implementation of the module, which complicates the implementation.

In AspectJ-like languages, aspects can be instantiated either as singleton or per-object(s); both impose limitations for implementing event processing applications. In the latter case, only the event stream that is produced by the objects to which an aspect instance is bound can be processed by the aspect instance; therefore, the implementations cannot cope with the dynamic changes in the number of event producers. In the former case, workaround must be provided to implement stateful event processing in which state information must be maintained based on individual or a certain group of event producers.

The stateless/stateful complex event processing can be implemented via advice code. Some languages [15] also offer history-based pointcut designators for this matter. However, the expression power of these pointcuts is limited by the adopted formalism. As it is extensively studied in the domain of runtime verification, different kinds of formalism with different expression power are usually needed for different kinds of stateful complex event processing.

The possibility to define the provided interface of aspects is restricted by the expression power of the language to define and select the join points that are activated within the advice code. AspectJ, for example, provides the pointcut designator *adviceexecution* for this matter. However, because advices are not named, it is not possible to distinguish between the advice in which the join points of interest are activated. Consequently, workaround methods must be provided to map the desired join points to method invocations and/or executions. Such workarounds cause the specification of module interfaces gets tangled with the implementation of the module, which complicates the implementations. Some languages such as EOS [12] unify the notion of aspects and objects; consequently, it is possible to select the join points of aspects in a similar way as objects.

With respect to the requirement *event-based composition*, such a composition can be achieved through join points and pointcut designators. In AspectJ-like languages, which support pointcut-based instantiation of aspects, the presence of an aspect instance depends on the presence of the base object to which the aspect instance is bound. Such a coupling does not exist in the languages that support explicit construction and deployment of aspects; for example in CaesarJ [11] and EOS [12]. In these languages, however, an aspect is limited to process the events that are produced by the objects on which it is deployed. One may provide code to dynamically deploy/undeploy aspects to cope with dynamic changes in the number of event producers; such code, however, scatters across and tangles with other concerns.

4.3 Stream-Processing Languages

Several different dedicated languages are introduced for event stream processing [4]. In this section, we evaluate Esper [3] as a representative of such languages. Esper is a component for complex event processing, which is available for Java. Complex event patterns, for example based on logical and temporal event correlation, can be expressed in an extension of SQL.

With respect to the requirement *event-representation*, the Esper language provides extensive support to define various kinds of events. For example, events can be defined as plain Java objects, XML, object-array, nested objects and hierarchical maps of objects. With respect to the requirement *event-based modularization of concerns*, a module is a plain text file in which the event processing queries are defined. Such queries are primitive statements in the Esper language, which are not represented as module abstractions. Therefore, the modularization requirements presented in Section 3 are not fulfilled. With respect to the requirement *event-based composition of concerns*, since there is no notion of modules, interface-level composition is not supported too. Instead, the composition is supported at the level of SQL queries to combine multiple queries with each other.

5. EVENT MODULES

In [8, 9] we introduced Event Composition Model, which offers a set of novel linguistic abstractions to effectively modularize and compose the concerns that typically appear in runtime verification techniques. In this section, we explain how these abstractions fulfill the requirements outlined in Section 3, and are consequently suitable to achieve modularity and loose coupling in the implementation of event processing applications.

As Figure 2 shows, at a high level of abstraction, Event Composition Model regards the execution environment as a set of **events** that may form a stream, and **event modules**. An event represents a state change of interest in the environment. Events are typed entities; an **event type** defines a set of **attributes** for the events. Two sets of attributes are supported: **static** and **dynamic**. The former includes the set of attributes whose values do not change and are known at the time an event is defined in the system. The latter defines the set of attributes whose values are known when an event is published during the execution of software.

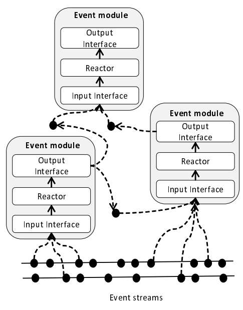

Figure 2: Event modules

An **event module** is a means to modularize a group of related events and the reactions to them. An event module is identifiable and referable by its unique **name**. It has a **required interface**, an implementation that is termed as **reactor**, and a **provided interface**.

The required interface of an event module specifies the set of events of interest to which the event module must react; the interface is activated when an event of interest occurs. The provided interface of an event module defines the set of events that are published by the event module to the environment. The implementation of an event module provides the functionality to process the events specified in the required interface of the event module, and to publish the events specified in the provided interface of the event module. The events provided by an event module can be selected further by other event modules; this facilitates the composition of event modules with each other.

In [8, 9], we introduced an early version of the EventReactor language, which implements Event Composition Model. The early version of EventReactor has three limitations. First, the instances of event modules are not accessible by their unique identity; this limits programming the instance-level composition of event modules with each other. Second, only implicit binding of event modules to each other is supported, which may lead to unnecessary runtime overhead. Third, the event selection is performed through a set of Prolog queries, which is not modularized. In this paper, we discuss a new implementation of event modules in the EventReactor language. Due to the space limit, we illustrate this via an implementation of our illustrative example and explain how the requirements outlined in Section 3 are fulfilled.

5.1 Event Representation

Event Composition Model does not fix the set of supported event types, attributes and events. To comply with this characteristic, EventReactor offers dedicated constructs to programmers to define new kinds of application and/or domain-specific event types, attributes and events. An event type is a data structure defining a set of attributes; there can be inheritance relation among event types.

```
eventtype Purchase extends EventType{
  dynamiccontext:
    long customerID;
    long productID;
}
eventtype FrequentPurchase extends Purchase{
  dynamiccontext:
    Purchase inner;
    long frequency;
}
eventtype OrdinaryPurchase extends Purchase{...}
eventtype MaxPurchase extends Purchase{...}
```

Listing 1: Specification of event types

`EventType` is the super type for the user-defined event types. To implement our illustrative example, we define the event types depicted in Listing 1. `Purchase` inherits from `EventType` and defines the attributes `customerID` and `productID`. Since the value of these attributes are not known until an event of this type is published, the attributes are defined as dynamic. `FrequentPurchase` and `OrdinaryPurchase` are two specializations of `Purchase`, which are used to distinguish between different kinds of purchases. These two event types define the attribute `inner` to keep a reference to the event that represents the original purchase request. The event type `FrequentPurchase` also defines the attribute `frequency` to keep the information about the frequency of purchase requests. `MaxPurchase` is another specialization, which is used to show the maximum amount of purchase in the 20 minutes. Due to the space limit, we do not show the specification of these event types.

Events can be published from software implemented in multiple languages, in the same way as explained in [8, 9]. To publish an event, it is necessary to initialize its dynamic attributes and inform the runtime environment of EventReactor of the event. EventReactor offers two APIs for this matter. In the first one, the information about the event is provided as a comma-separated list of attributes and their values. This API is useful if event producers are not Java programs. The second API is useful if the events are published from a Java program. To utilize this API, EventReactor generates Java classes from the specification of event types. One must define instances of the event types in a similar way as pure Java objects, initialize their dynamic attributes and publish the event by invoking the API. Due to the space limit, we do not represent the event publishing code.

5.2 Event-Based Modularization of Concerns

As Figure 2 shows, individual event processing agents and/or event consumers can be modularized as individual event modules. Event modules fulfill the modularity requirements outlined in Section 3; they are referrable via their unique names, have event-based interfaces, and can express stateless/stateful event processing logics via their reactors.

```
eventmodule CountEvents{
 requires{ Purchase p_event; Reserve r_event; Cancel c_event;}
 provides{}
 reactor{ if (Shopping.computeElapsedTime()< 20){
        if (p_event) p_counter++;
        else if (r_event) r_counter++;
          else if (c_event) c_counter++;
        }
      else{
        Shopping.log(p_counter, r_counter, c_counter);
        Shopping.reset(p_counter, r_counter, c_counter);
      }
 }
 variables{ long p_counter, r_counter, c_counter;}
}
eventmodule FilterEvents{
 requires{ Purchase p_event;}
 provides{ OrdinaryPurchase o_event; FrequentPurchase f_event;}
 reactor{
  frequency = Shopping.getPurchaseFrequency(p_event, 10);
  if (frequency > 30){
    f_event.inner = p_event;
    f_event.frequency = frequency;
    publish f_event;}
  else{
    o_event.inner = p_event; publish o_event;}
  }
  variables{ long frequency;}
}
eventmodule ComputeMax{
 requires{ Purchase p_event;}
 provides{ MaxPurchase mp_event;}
 reactor{
  if (Shopping.computeElapsedTime()< 20){
    maxpurchase = Shopping.max(p_event.amount, maxpurchase);}
  else{ mp_event.max = maxpurchase; publish mp_event;}
  }
 variables{ long maxpurchase;}
}
eventmodule LogInfo{
 requires{ MaxPurchase event;}
 provides{}
 reactor{ Shopping.log(event.max);}
}
```

Listing 2: Modularizing event processing agents

Listing 2 shows an excerpt of the event modules that modularize the event processing agents of our running example. The event module `CountEvents` implements the stateful functionality to count the number of events, which represent the requests to purchase a product, reserve a product or cancel a purchase, within the last 20 minutes. `CountEvents` receives these events via its required interface; the event module does not provide any event. Within this event module, three counter variables are defined, which are updated by the reactor part whenever there is an event in the corresponding interface. The value of these counters are printed and restarted each 20 minutes. We provide the helper Java class `Shopping`, which implements the necessary statistical functions. For the sake of brevity, we do not show the implementation of this class.

The event module `FilterEvents` implements the stateful functionality to separate the purchase events into ordinary and frequent ones. For this matter, it receives an event of the type `Purchase` in its required interface, and provides the events `o_event` and `f_event` of the types `OrdinaryPurchase` and `FrequentPurchase`, respectively. The reactor part computes the frequency of the shopping for each customer in the last 10 days. If this is above the threshold, the event `f_event` is published; otherwise, the event `o_event` is published. Before publishing these events, necessary values are assigned to their attributes.

The event module `ComputeMax` implements the stateful functionality to compute the maximum amount of ordinary and frequent purchases in the last 20 minutes. To be able to reuse this event module for both ordinary and frequent purchases, the required interface of this event module receives an event of the type `Purchase`, which is the super type of `OrdinaryPurchase` and `FrequentPurchase`. An event of the type `MaxPurchase`, which keeps the maximum amount of purchase in its attribute `max`, is published by this event module. The event module `LogInfo` implements the stateless functionality to log the maximum amount of purchase.

5.3 Event-Based Composition of Concerns

To utilize event modules, they must be instantiated and their interfaces must be bound to the desired events. To achieve loose coupling in the compositions, the new version of EventReactor facilitates a) explicit instantiation of event modules so that instance-level composition of event modules with event streams can be achieved; b) separating composition specifications from the specification of event modules; c) implicit and explicit binding of events to the interfaces of event modules; and d) polymorphic binding of events to interfaces.

In the implicit binding, whenever an event is published to the runtime environment of EventReactor, the event is bound to the required interfaces of event modules, whose type matches the type of the event. In the explicit binding, programmers specify the bindings between the required interfaces and provided interfaces of event modules. Implicit binding creates looser coupling among event modules, and among event modules and event producers; the absence or presence of an event producer does not influence the event modules. Implicit binding may come with the price of extra runtime overhead. Therefore, when needed, explicit binding can be adopted. The explicit binding offered by EventReactor can still lead to some degree of loose coupling, because the interfaces of event modules are event-based and the binding takes place at the interface level separately from the event modules.

In both implicit and explicit bindings, the type of events is polymorphically checked against the types of required interfaces; this facilitates keeping interfaces loosely coupled to specific event types. In both kinds of binding, since the required interface of event modules can be bound to multiple events that may come from multiple sources, the event modules can effectively modularize crosscutting event processing logics.

Listing 3 shows an excerpt of the composition code that is provided for our example. Here, we define one instance of `CountEvents`, `FilterEvents` and `LogInfo`. To separately compute the maximum amount of purchase for ordinary and frequent purchases, two instances of the event module ComputeMax are defined. Considering Figure 1, the event modules CountEvents and FilterEvents are the first event processing agents in the chain, which receive the events directly from the event streams. To achieve loose coupling between the event modules and the event producers, we leave the required interfaces of ce and fe unbound, so that EventReactor performs implicit binding based on the type of the published events. Lines 8–9 show an example of polymorphic explicit binding, in which the provided interfaces of fe are bound to the required interface p_event of cpmaxOrdinary and cpmaxFrequent, respectively. Lines 10–11 show the binding to the required interface event of li, so that logging can be performed when the specified events are provided by cpmaxOrdinary and cpmaxFrequent.

If an event is processed by multiple event modules, the processing order can be specified via the operator precede. In our example, since the events of the type Purchase are processed by both ce and fe, we specify that ce must process the events first.

```
1  composition {
2    CountEvents ce;
3    FilterEvents fe;
4    ComputeMax cpmaxOrdinary;
5    ComputeMax cpmaxFrequent;
6    LogInfo li;
7
8    bind (fe.o_event, cpmaxOrdinary.p_event);
9    bind (fe.f_event, cpmaxFrequent.p_event);
10   bind (cpmaxOrdinary.mp_event, li.event);
11   bind (cpmaxFrequent.mp_event, li.event);
12
13   precede (ce, fe);
14 }
```

Listing 3: Composing event modules

Assume for example that at runtime, events of the type Purchase are published. EventReactor matches each event against the required interfaces of the instantiated event modules. In our example, the event matches the required interface p_event of ce, fe, cpmaxOrdinary and cpmaxFrequent. Since the interfaces of the latter two event modules are explicitly bound, EventReactor only binds the event to the interface p_event of ce and fe. As it is specified, ce must process the event first; after this event module finishes the event processing, fe starts the event processing. The chain of event processing continues according to the specified explicit bindings among the event modules.

6. CONCLUSION AND FUTURE WORK

We discussed that to be able to effectively cope with the complexity of event processing applications, separation of concerns and loose coupling among the concerns must be achieved in the implementations. Event-based composition is known to facilitate loose coupling in compositions. We illustrated that an effective event-based composition requires an event-based modularization of concerns in which modules interfaces are defined in terms of events. We defined a set of language requirements for event-based modularization and composition of concerns, and explained that the current languages significantly fall short to fulfill the requirements. A new implementation of event modules in the EventReactor language was explained and its suitability to separate the concerns and compose them with each other in a loose manner was illustrated.

There are several languages that provide dedicated features for event processing, and there is a need for a comparison framework to evaluate them with respect to modularity and compose-ability of implementations. This paper took an initial step towards this. As future work, we would like to extend our set of language requirements, for example to include parallelism, and would like to extend our evaluation with a larger set of languages.

Various kinds of architectural patterns (e.g. pipe and filter and blackboard) can be adopted for event processing applications. As future work, we would like to evaluate the suitability of event modules for implementing such patterns. We would also like to formally specify the semantics of event modules in processing events.

Acknowledgments

The author would like to thank Prof. Mehmet Aksit and Prof. Shmuel Katz for the discussions on the earlier drafts of this paper.

7. REFERENCES

[1] M. Akşit. Separation and Composition of Concerns. *ACM Computing Surveys*, 28, 1996.

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

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

[4] O. Etzion and P. Niblett. *Event Processing in Action*. Manning, 2010.

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

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

[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP' 97*. Springer-Verlag.

[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 Modules: Modularizing Domain-Specific Crosscutting RV Concerns. In *TAOSD (to appear)*, LNCS. 2013.

[10] S. Malakuti, C. Bockisch, and M. Aksit. Applying the Composition Filter Model for Runtime Verification of Multiple-Language Software. In *ISSRE '09*, 2009.

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

[12] H. Rajan and K. Sullivan. Eos: Instance-Level Aspects for Integrated System Design. In *ESEC/FSE-11*, 2003.

[13] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):14:1–14:42, May 2009.

[14] F. Steimann, T. Pawlitzki, S. Apel, and C. Kästner. Types and Modularity for Implicit Invocation with Implicit Announcement. *ACM Trans. Softw. Eng. Methodol.*, 20:1:1–1:43, July 2010.

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