View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Technische Universität Dresden: Qucosa

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik

TECHNISCHE BERICHTE
TECHNICAL REPORTS

ISSN 1430-211XX

TUD-FI15-03-August 2015

Somayeh Malakuti
Software Technology group

An Overview of Language Support for Modular Event-driven Programming

TechnischeUniversität Dresden
Nutzerberatung CD
01062 Dresden
Germany
http://tu-dresden.de/cd

DRESDEN concept
Exzellenz aus Wissenschaft und Kultur

An Overview of Language Support for Modular Event-driven Programming

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

August 6, 2015

Abstract

Nowadays, event processing is becoming the backbone of many applications. Therefore, it is necessary to provide suitable abstractions to properly modularize the concerns that appear in event-driven applications. We identify four categories of languages that support event-driven programming, and identify their shortcomings in achieving modularity in the implementation of applications. We propose gummy modules and their implementation in the GummyJ language as a solution. Gummy modules have well-defined event-based interfaces, and can have a primitive or a composite structure. Composite gummy modules are means to group a set of correlated event processing concerns and restrict the visibility of events among them. We provide an example usage of gummy modules, and discuss their event processing semantics.

1 Introduction

An *event* is usually defined as something that happens, especially something important, interesting or unusual. Event-driven programming is a paradigm in which the flow of programs is determined by events. Nowadays, event-driven programming is finding its way in many application domains such as runtime verification techniques [30], self-adaptive software systems [12], IoT applications, and various monitoring systems such as traffic monitoring.

As for any other application, it is necessary to properly separate and modularize the concerns that appear in event-driven applications. In this report, we outline a set of requirements that a language must fulfill for modular implementation of event-driven applications. We identify four categories of languages that support event-driven programming, and identify their shortcomings with respect to the outlined requirements. The shortcomings can be summarized as: (a) incomplete event-based interfaces for modules, (b) limited support for

content-based filtering of events, (c) tight couplings in the composition of modules, and (d) limited support for defining the visibility of events.

To overcome these shortcomings, we propose *gummy modules* as novel abstractions to separate and modularize event processing concerns. Gummy modules, which have well-defined event-based interfaces, can have a primitive or a composite structure. A composite gummy module is a means to group a set of correlated event processing concerns. Thus it is a way to restrict the visibility of events among the correlated event processing concerns. Gummy modules are parts of the larger concept of *event-based modularization* [26], which is the successor of current modularization mechanisms dedicated for event-driven applications.

In this report, we extend our previous work [25, 27, 26] in the following ways. (a) We identify four categories of languages support for event-driven programming. (b) We evaluate the shortcomings of a large set of languages and techniques in modularizing event-driven applications, and outline a set of requirements to overcome these shortcomings. (c) We explain the concept of composite gummy modules and their use for restricting the visibility of events. (d) We explain an implementation of composite gummy modules in GummyJ, and explain its integration with Java applications.

The rest of this report is organized as follows: Section 2 provides background information about event-driven applications, and explains our illustrative example. Section 3 outlines a set of requirements for modular implementation of event-driven applications. Section 4 evaluates the suitability of current languages and outlines a set of requirements for implementing and modularizing such applications. Sections 5 and 6 explain gummy modules and their implementation in GummyJ, respectively. Section 7 discusses the event processing semantics of the GummyJ language. Section 8 discusses the suitability of gummy modules in improving the modularity of implementations. Section 9 discusses the related work. Section 10 explains the evolution to current modularization mechanisms to event-based modularization, and Section 11 outlines conclusion and future work.

2 Background

An *event* is usually defined as something that happens, especially something important, interesting or unusual. Event-driven programming is a paradigm in which the flow of programs is determined by events. In an event-driven application, various software and/or hardware entities can produce and consume events. There are the so-called *event processing concerns (EPC)* (also known as event processing agents [14]), which are software entities mediating between event producers and event consumers to process and reason about the events. Naturally, EPCs also produce and consume events.

The general belief is that event-driven programming helps to keep event producers and consumers decoupled from each other. Three types of decoupling are distinguished in the literature [17]: space, time and synchronization. The space

decoupling means that event producers and consumers do not keep a reference to each other; hence, they are unaware of each other. The time decoupling means that event producers and consumers do not have to be active at the same time; hence, they may have different lifetime. The synchronization decoupling means that event producers do not get blocked while publishing events, and consumers can asynchronously get notified of new events.

Traditionally, event-driven programming is considered suitable for handling the interactions of users with GUI and dealing with hardware interrupts. However, event processing is becoming the backbone of many today's applications such as runtime verification techniques [30], self-adaptive software systems [12], IoT applications, and various monitoring systems such as traffic monitoring and stock market monitoring systems.

We take a runtime verification application as our case study. Assume for example that there is a remote file management application. The requests for managing folders and files are issued as events from client applications. An example set of events are shown in Figure 1, where a user logs in, opens a folder, opens a file, etc.

Each user is assigned a unique identifier, and the users are categorized into *premium* and *silver* users. Initially, the server-side objects receive events and directly handle them, but the application must evolve with the following set of EPCs, which are also shown in Figure 1.

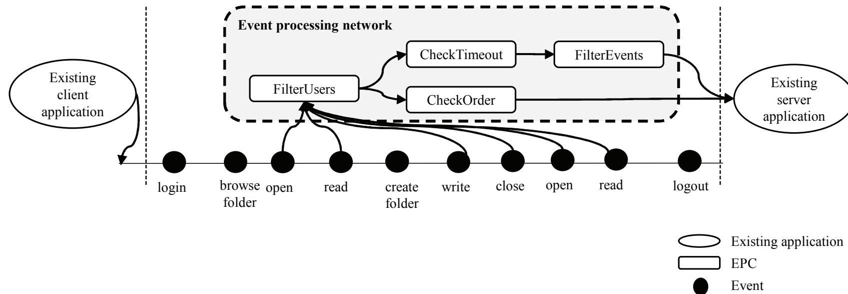

Figure 1: The example application

Evolution (1): An EPC named as *CheckTimeout* must be defined to ensure that a file is not left open for more than T seconds without any read or write requests on it. Otherwise, an event indicating a timeout error must be published.

Evolution (2): The EPC *CheckTimeout* must only be applied to the premium users. An EPC named as *FilterUsers* must be defined to implement the functionality for selecting the events that are published by premium users.

Evolution (3): The first four times that a user keeps a file open for an excessive amount of time can be tolerated each day. The timeout event must be published on the fifth occurrence in a day. A user for whom a timeout error is issued can no longer access the file until next day.

Evolution (4): An EPC named as *CheckOrder* must be defined to check whether the events for accessing a file are issued in a correct order. A correct order is defined as one *open* event, followed by zero or more *read* and/or *write* events, and finally one *close* event.

A set of correlated event processing concerns forms the so-called *event processing network* [14]. For example as Figure 1 shows, we group the EPCs as one event processing network named as *PremiumUsersEPCs*.

3 Requirements for Modular Event-driven Programming

Event-driven applications may be complex and subject to frequent evolution demands. Therefore, as for any other application, it is necessary to properly separate and modularize the concerns that appear in such applications.

We observe that at least three different kinds of concerns appear in event-driven applications. (a) The so-called base concerns, which implement the base functionalities of the applications; for example, accessing files and folders. (b) The concerns that crosscut [24] the base concerns; for example, authentication and persistence crosscutting concerns. (c) The EPCs, which implement the functionality to consume, process and produce events.

In the literature, procedural, object-oriented, and aspect-oriented [24] mechanisms are proposed to modularize base and crosscutting concerns in applications. To be able to cope with the complexity of event-driven applications, we claim that a language must also facilitate modularizing individual EPCs, and must offer means to compose the modularized EPCs into event processing networks. The following requirements must be fulfilled in this regard:

- Event processing modules must have well-defined event-based interfaces. Since not all events are of the interest for a module, the interfaces must specify the criteria for filtering the events of interest. In addition, they must specify the set of events that are produced by the modules. For example in Figure 1, *FilterUsers* is only interested in file-related events, where *CheckOrder* and *CheckTimeout* are only interested in the events provided by *FilterUsers*. Specifying both produced and consumed events paves the way to check various typing rules, for example, checking covariant return types in inheritance hierarchies.

- Events are regarded as data in motion; they enter a module and may leave it in their original or in a transformed form. Therefore, in addition to their internal states, event processing modules must facilitate restricting the visibility of their events. The visibility rules apply to events as data, and their motion from specific producers to consumers.

- Event processing modules must be composed with each other at the interface level. In the spirit of composite objects, a language must facilitate

composing a set of correlated event processing modules into an event processing network. The space, time, and synchronization decoupling must be preserved in compositions.

4 An Evaluation of Programming Languages

The current trend in language support for event-driven programming is to adopt an existing language and extend it with some event processing features. We classify these extensions into four categories: (a) languages with an event notification/delegation mechanism [31], (b) languages with an event quantification mechanism [32, 19, 27, 30, 25], (c) languages with stream/complex event processing [1, 35, 16], and (d) languages with a publish/subscribe mechanism [15, 9, 20]. Each category proposes a different style of event-driven programming, which addresses the requirements outlined in Section 3 differently.

Figure 2 summarizes the results of our evaluation. Here, the symbols +, × and - mean that a requirement is supported, partially supported or not supported, respectively.[1]

Category	Language	Modules with event-based interfaces	Event visibility	Interface-level Composition	Decoupling		
					Space	Time	Synchronization
Languages with an event delegation mechanism	C#-like Languages	×	+	×	-	-	×
Languages with an event quantification mechanism	Ptolemy	×	-	×	×	-	-
	EScala	×	+	×	×	-	-
	EventReactor	+	-	+	×	-	-
	JavaMoP	×	+	×	×	-	-
	GummyJ 1.0	×	-	×	+	+	+
	GummyJ 2.0	+	+	+	+	+	+
Languages with a stream/complex event processing mechanism	Esper	-	-	-	×	×	+
	StreamIT	-	+	-	+	×	+
	EventJava	×	+	×	-	-	+
Languages with a publish/subscribe mechanism	$Java_{PS}$	-	-	-	+	+	+
	ECO	-	-	-	+	+	+
	CORBA	-	+	-	+	+	+

Figure 2: The summary of the evaluation

[1]We assume there is a time coupling if the absence of event consumers (producers) when publishing (receiving) events results in errors. Hence, when there is a synchronization coupling or explicit references among event producers and consumers, there will be a time coupling too. We do not consider advanced cases such as event replication when one party is absent.

4.1 Languages with an Event Delegation Mechanism

Some mainstream languages such as C#, JavaScript and Python offer dedicated constructs, which are a simple implementation of the Observer pattern, to define EPCs. Listing 1 shows a code snippet for our illustrative example in C#. `EventArgs` is a predefined type in C# to represent event types. Line 2 defines the event type `FileEvent` as a subclass to represent file-related events. As line 3 shows, concrete operations on files are defined as subclasses of `FileEvent`.

```
1  //Defining event types
2  public class FileEvent:EventArgs {private string _user; private string _filename};
3  public class OpenEvent: FileEvent{...}
4  ...
5  //Defining event delegates
6  public delegate void FileEventHandler(object sender, FileEvent e);
7  ...
8  //Defining EPCs
9  class FilterUsers {
10     public event FileEventHandler file_event;
11     public void filter(object sender, FileEvent e){
12      if ((e is OpenEvent || e is ReadEvent || e is WriteEvent || e is CloseEvent) &&
13           e.User == 'premium'){
14             if (file_event != null) file_event(this, e); }
15     }
16 }
17 class CheckTimeout{...}
18 class CheckOrder{...}
19 class FilterEvents {...}
20 class PremiumUsersEPCs{
21     public FilterUsers userselection;
22     private CheckTimeout timerchecker;
23     private CheckOrder orderchecker;
24     public FilterEvents timeouterror;
25     public PremiumUsersEPCs(){
26         userselection = new FilterUsers(this);
27         timerchecker = new CheckTimeout();
28         orderchecker = new CheckOrder();
29         timeouterror = new FilterEvents(this);
30         userselection.file_event += new FileEventHandler(timerchecker.check);
31         ...
32     }
33  }
```

Listing 1: An example code in C#

In C#, the methods that process events are invoked through delegates. Line 6 defines an event delegate for the events of the type `FileEvent`. To maximize reuse, we define each EPC of our example as a separate class. Lines 9–16 implement the functionality for the EPC *FilterUsers* of our example. Here, the

method `filter` is executed when an event of the type `FileEvent` is received. There can be many subclasses of `FileEvent` in the application, and we are only interested in the events of the types `OpenEvent`, `ReadEvent`, `WriteEvent` and `CloseEvent`. The check in line 12 is to filter the events whose type is of interest. The check in line 13 ensures that the event is published by a premium user. If these conditions hold, the event is published further in line 14.

Other EPCs are defined likewise in separate classes in lines 17–19. In C#, event producers and consumers must explicitly be bound to each other. The class `PremiumUsersEPCs` in lines 20–33 defines an event processing network by grouping the previously defined EPCs, instantiating them and bounding them together. The expression in line 30, for example, indicates that `CheckTimeout` must process the event `file_event` that are published by `FilterUsers`.

This style of event-driven programming has the following shortcomings with respect to the requirements outlined in Section 3.

Firstly, as line 11 in Listing 1 shows, the signature of methods only specifies the events that are consumed by the methods. The events produced by the methods must be specified at the level of objects; for example as shown in line 10. Consequently, method-level type checking such as checking for covariant return types cannot be applied to the the events produced by the methods.

Secondly, interface-level composition is supported via a limited kind of type-based event filtering. Consequently, the code for filtering events based on other attributes must be defined within the body of methods. Such code tangles with and possibly scatters across the core functionality of the methods. This is shown for example in line 12 of Listing 1, where we would like to only match specific subtypes of `FileEvent`. Alternatively, one may define individual methods for matching individual event types. However, this causes the number of methods to grow in proportion to the number of event types, and makes programs verbose.

Thirdly, as line 11 of Listing 1 shows, a reference to event producers is sent to event consumers. As line 30 of Listing 1 shows, event producers and consumers must explicitly be bound to each other to maintain the flow of events among them. On one hand, such explicit bindings helps to limit the visibility of events in C#. On the other hand they result in space and time couplings among producers and consumers, and make applications error-prone. For example, if the check in line 14 is mistakenly omitted, an exception will be raised when there is no consumer registered for the produced event. Or, in an application with varying number of event producers and consumers, we have to update the bindings every time the number of event producers and consumers changes.

Fourthly, there can be a synchronization coupling among event producers and consumers because events are published via method invocations, which are by default synchronous. The synchronization decoupling can be programmed via multi-threading in C#.

4.2 Languages with an Event Quantification Mechanism

In aspect-oriented (AO) languages [24], join points are means to represent a state change of interest in the execution of programs. Pointcut designators

are means to select the joint points of interest. Events and join points can be considered analogous [34]. Therefore, one may consider adopting AO languages to benefit from pointcut designators to query the events of interest.

There are various languages that explicitly support the notion of events, and offer an event quantification mechanism similar to pointcut designators [32, 19, 27, 23, 25, 30]. To evaluate these languages we adopt Ptolemy [32] as a representative.

Listing 2 shows a code snippet for our illustrative example in Ptolemy. The EPC *FilterUsers* of our illustrative example is implemented in lines 1–8. The expression in line 7 specifies that when an event of the type `FileEvent` or any of its subtypes is published to the Ptolemy runtime, the method `filter` must process the event. Within the body of this method, it is checked whether the event has one of our desired types. If it is the case and the publisher of the event is a premium user, the event is forwarded to the next event consumer in the chain, if any, via `next.invoke()`. Lines 9–12 define another EPC of our example. Lines 16–19 show that the instances of the class `FilterUsers` and `CheckOrder` can be registered in the Ptolemy runtime via the statement `register`.

```
1  public class FilterUsers{
2    public void filter(FileEvent next) {
3          if (next instanceof OpenEvent || next instanceof ReadEvent
4          || next instanceof WriteEvent || next instanceof CloseEvent)
5             if (next.role() == "premium") next.invoke();
6    }
7    when FileEvent do filter;
8  }
9  public class CheckOrder{
10   public void check(FileEvent next) { ... next.invoke(); }
11   when FileEvent do check;
12 }
13 public class Main{
14 public void init(){
15   ...
16   FilterUsers userselection = new FilterUsers();
17   CheckOrder orderchecker = new CheckOrder();
18   register (userselection);
19   register (orderchecker);
20   ...
21 }
22 }
```

Listing 2: An example code in Ptolemy

With respect to the requirements in Section 3, we observe the following problems in this style of event-driven programming.

Firstly, as for C#, the signature of methods only specifies the set of events that are consumed by them. Secondly a limited kind of type-based event filtering

is supported. As shown in lines 3–4 of Listing 2, more complex filtering criteria that are based on the content or type of events must be expressed in the body of methods; such code tangles with the core functionality of the methods.

Thirdly, Ptolemy does not offer means to limit the visibility of events; events are globally visible to all Ptolemy objects. As a workaround, one may provide code for filtering out irrelevant events. Such code must be defined within the body of the methods that process events, tangled with the core functionality of the methods.

Fourthly, since events are published in a synchronous manner, there are time and synchronization couplings among event producers and consumers. Nevertheless, since event consumers are instantiated and registered in the runtime environment of Ptolemy, they react to the events obliviously of the actual event producers. Therefore, there is no space coupling among the event producers and consumers.

Fifthly, in the spirit of the *around* advice in AO languages, the flow of events must explicitly be maintained among event consumers by invoking the next consumer in the chain. For example in Listing 2, the instance of `FilterUsers` processes file events first, and when it invokes the next object in the chain via `next.invoke()`, the instance of `CheckOrder` will process the events. This approach creates time, space and synchronization couplings among event consumers. Consequently, applications become error-prone and hard to evolve because new event consumers cannot easily be added to the applications without explicitly maintaining the flow of events among them.

Other languages in this category are EScala [19], EventReactor [27], JavaMoP [30], and GummyJ [25]. EScala [19] extends the interface of objects with declarative events, which represent specific state changes in the objects. In addition to supporting events in the same way as C#, EScala supports implicit events, which correspond to the method-based join points in AO languages. These events can be selected and composed by the available pointcut predicates. The visibility of events can be controlled in a similar way as state variables. As for C#, there are time and synchronization couplings among event producers and consumers. Since event producers can keep a reference to event consumers, there is a space coupling in the producer side. EScala objects have incomplete event-based interfaces because they only define the set of events that they produce.

EventReactor [27] facilitates modular implementation of EPCs via offering a dedicated module abstraction named as *event modules*. These modules have event-based interfaces, but do not facilitate restrict the visibility of events. Besides, there are synchronization and time couplings among event producers and consumers because events are announced via synchronous method calls to the language runtime. There can be a scope coupling at the consumer side, because via event attributes event consumers can have a reference to event producers.

JavaMoP [30] is a domain-specific language based on AspectJ for runtime verification of a set of correlated events. It considers join points as events, and adopts pointcut expressions as means to specify the events of interest that are consumed by aspects. However, it is not possible to specify the the set of events

that are published by aspects. The visibility of events is limited to a base object and the aspects bound to it. There is space coupling in the aspects side, because aspects they can refer to base objects; there are also time and synchronization couplings.

We proposed the GummyJ 1.0 language [25], which facilitates modularizing EPCs via dedicated module abstractions named as *emergent gummy module*. These modules encapsulate their construction and destruction semantics, so they are decoupled in time and space to event producers. Events are published in an asynchronous manner, so there is also synchronization decoupling. Modules specify their set of required events and the criteria to filter the events, but not their provided events. Besides, there is no support for limiting the visibility of events.

4.3 Languages with Complex Event Processing

A more advanced form of event quantification is provided by stream/complex event processing languages [1, 35, 16]. These languages usually offer mechanisms for detecting certain patterns in events, aggregating and transforming events, modeling event hierarchies, detecting relationships (such as causality, membership or timing) between events, etc. In this section we focus on Esper [1] as a representative of these languages. Listing 3 is an example code snippet.

Lines 1–14 implement the functionality for the EPC *FilterUsers* of our example. Esper offers the concept of named windows, which are data windows that can be inserted into, deleted from, and queried by one or more statements. To filter out the events that are published by premium users, the expression in line 8 of Listing 3 creates a window named as `PremiumUsers`. The expression in lines 10–11 select relevant events and inserts them into this window.

Lines 15–27 implement the functionality for the EPC *CheckOrder* of our example for the premium users. It selects events from the previously-defined named window `PremiumUser`. Esper facilitates defining the temporal order of events using regular expression.

With respect to our requirements, firstly, the signature of methods do not specify the set of events that are processed or produced by them. Consequently, compositions cannot take place at the level of interfaces; instead they must be done at the the level of SQL statements within the body of methods. For example to compose `FilterUsers` with `CheckOrder`, we define the named window `PremiumUsers` in line 10 of Listing 3, which is referred to in the SQL statement in line 20.

Secondly, even if named windows are adopted, events are globally visible by default. Workarounds may be provided by filtering out irrelevant events via SQL statements, if possible at all. This increases the complexity of the statements, and reduces the reusability of the implementations if they must be reused in different scopes. For example, `CheckOrder` is hardcoded to work with the named window `PremiumUsers`; this however does not mean that the events in this window are only visible to `CheckOrder`.

Thirdly, event producers publish events the Esper runtime, which provides them to event consumers. So, there is time, space and synchronization decoupling among event producers and consumers. However, since event consumers are composed with each other via shared tables/data windows in their SQL statements, there is a time coupling among them. If we regard those shared tables/data windows as implicit references, there is a space coupling among event consumers too.

```
1  public class FilterUsers{
2          public FilterUsers(string user, final EPServiceProvider
3          epService, final AlertListener fileAlterListener)
4          {
5            if (user == "Premium"){
6               EPStatement namedWindowStmt =
7               epService.getEPCdministrator().createEPL(
8               "create window PremiumUsers.win:keepall() (userID String, type String)");
9                  EPStatement insertWindow = admin.createEPL(
10                 "insert into PremiumUsers select userID, type from
11                  FileEvent where userID = '"+user+"'");
12           }
13         }
14 }
15 public class CheckOrder {
16        public void CheckOrder(final EPServiceProvider epService, final
17        AlertListener fileAlterListener){
18           EPStatement namedWindowStmt = epService.getEPCdministrator().
19              createEPL(
20                 "select * from PremiumUsers
21                       match_recognize (measures o[0].userID as user
22                       pattern ((open (read | write )* close)*)
23                       define open as open.type = 'open', close as close.type = 'close',
24                       read as read.type = 'read', write as write.type = 'write')"
25                  );
26         }
27 }
```

Listing 3: An example code in Esper and Java

StreamIT [35] is a dataflow language targeting fine-grained highly parallel stream applications and providing a highly optimizing native compiler. It defines the concept of *filters*, which are special Java classes whose methods implement event processing code. Filters can be composed with each other via channels in a sequential, parallel or feedback-loop way. Channels are FIFO queues, which also facilitate limiting the visibility of events between filters that enqueue/dequeue events in/from the channels. Filters are decoupled and run independently, however, they do not have event-based interfaces.

In EventJava [16] an application event type is implicitly defined by declaring an event method, which is an asynchronous method. The signature of such

methods may define complex criteria to filter events based on their attributes and/or correlations. With respect to our requirements, the method signatures do not specify the set of published events, and there are time and space couplings among event producers and consumers due to explicit invocations on consumers to publish events. Publishing events via invocation is a means to limit the visibility of events among specific event producers and consumers.

4.4 Languages with a Publish/Subscribe Mechanism

The publish/subscribe paradigm is widely accepted for developing applications that require one-to-many and many-to-one style of communication. Various commercial middleware and programming languages are proposed to support this paradigm [15, 9, 20]. Listing 4 shows a code snippet for our *CheckOrder* EPC in the Java$_{PS}$ language [15]. Lines 1–10 define an event consumer (i.e. subscription in Java$_{PS}$ terminology), which selects events of the type `FileEvent`. The Boolean expression in lines 4–5 specifies that only events of the the specified sub-types are of the interest. The code for checking the order of events must be provided in Java in lines 6–8. Lines 11–16 implement an event producer, which instantiates an event of the type `OpenFile` and publishes it to the Java$_{PS}$ runtime.

```
1  public class CheckOrder {
2        public void CheckOrder(){
3        Subscription fileSubscription = subscribe (FileEvent event){
4                return (event instanceof OpenEvent || event instanceof ReadEvent
5                || event instanceof WriteEvent || event instanceof CloseEvent)}
6          {
7            //check order functionality
8          };
9          }
10 }
11 public class Producer{
12   public void produce (){
13     OpenFile open_event = new OpenFile("F1.txt", "premium");
14     publish open_event;
15   }
16 }
```

Listing 4: An example code in Java$_{PS}$

ECO [20] offers an API in C++ for programming publish/subscribe systems, which support asynchronous distribution of events from producers to all consumers that are interested in those events. Commercial middleware such as CORBA [9] also offer APIs for event-driven programming. Using these APIs, one can define communication channels between event producers and consumers, and plug event filtering code into the middleware.

The main focus in this style of event-driven programming is to increase the decoupling of event producers and consumers, publishing and receiving events in

an asynchronous way. As for the languages in the previous section, supporting modules with well-defined event-based interfaces, and composing modules at the interface level are not the focus.

4.5 Summary of the Evaluation

In this section, we identified four flavors of supporting event-driven programming in languages. Figure 3 depicts the degrees of modularity and decoupling in current languages. As explained before, there are some attempts to support modular implementation of EPCs in the first two categories of languages. However, they have various shortcomings in this regard. An interesting insight is that although decoupling is one of the distinguishing characteristics of event-driven applications, these languages put less emphasize on achieving it. This decision can be justified that small-scale event-driven applications with a fixed number of event producers and consumers (e.g. GUI applications) do not require to have all three sorts of decoupling.

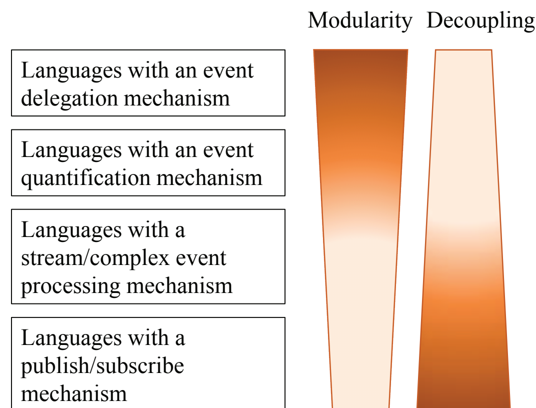

Figure 3: The degrees of modularity and decoupling in current languages

As we move towards the fourth category, the languages become more suitable for larger-scale applications that need to deal with numerous decoupled event producers and consumers. The languages in the third and fourth categories mainly focus on increasing the decoupling in applications and providing means to process large number of events. We believe that increasing the reusability and maintainability of application via proper separation and modularization of concerns are also important factors that must be taken into account for large-scale applications.

5 Gummy Modules

In this section we explain **gummy modules**[2], which are means to modularly define EPCs in programs. Figure 4 represents an BNF description of gummy modules; note that this is not the syntax of a particular language in which this concept is implemented.

At a high level of abstraction, we assume that *Program* consists of a set of typed events, primitive and/or composite gummy modules, and ordinary application classes. An *EventType* has a unique identifier, and defines a set of attributes, which are means to represent necessary contextual information about events.

Gummy modules may have a primitive structure. As the expression *PrimitiveGummyModule* shows, a primitive gummy module is recognized by its unique identifier, and it may define a set of local variables, a set of ordinary methods similarly to conventional objects. These are not accessible outside the module. It may also define a set of event-based methods. Such methods have a unique identifier, filter a set of events via their *PrimitiveSelector*, process them via their internal *PrimitiveReactor*, and may produce new events of the *PrimitivePublisher* type.

Primitive selectors are predicates over event attributes to filter the events of interest. Primitive reactors are pieces of programs in a language, which process the events that are received from the selectors and may publish new events. Primitive publishers specify the type of events that can be published outside the gummy module.

Gummy modules may also have a composite structure. As the expression *CompositeGummyModule* shows, such modules may refer to other gummy modules as their selector, reactor and/or publisher.

As for local variables, both primitive and composite gummy modules may define a set of *SharedVariable*s, which are means to share data among a composite gummy modules and its constituent modules.

Figure 5 provides a schematic representation for two example gummy modules. At the top of the figure, a primitive gummy module is shown, which has three event-based methods. At the bottom, a composite gummy module is shown. Composite gummy modules are means to group a set of correlated gummy modules, and facilitate defining and modularizing complex semantics for selecting, processing and publishing events. In addition, composite gummy modules facilitate limiting the visibility of events. Here, the events filtered by the selectors of a composite gummy module are only visible within the module by its reactors and publishers. The events published by the reactors are not visible outside the module unless they are sent out by the publishers of the composite gummy module.

As shown in the figure, the selector parts of composite selectors are recursively exported as the selectors of the enclosing composite gummy module;

[2]The number of event producers from which gummy modules receive events may increase or decrease at runtime. Therefore, the composition of gummy modules with event producers has elastic nature. The term 'gummy' emphasizes this elasticity.

- Program ::= (<EventType> | <PrimitiveGummyModule> | <CompositeGummyModule> | <Class>)*
- EventType ::= <Identifier> <Attribute>*
- Attribute ::= <Identifier><Type>
- PrimitiveGummyModule ::= <Identifier> (<Variable> | <Method> | <EventMethod> | <SharedVariable>)*
- CompositeGummyModule ::= <Identifier> (<Variable> | <Method> | <SharedVariable> |
  <CompositeSelector> | <CompositeReactor> | <CompositePublisher>)*
- EventMethod ::= <Identifier><PrimitiveSelector><PrimitiveReactor><PrimitivePublisher>
- PrimitiveSelector ::= $q \in Q$ where Q is a set of acceptable logical predicates over event attributes
- PrimitiveReactor ::= $s \in S$ where S is the set of supported statements in the language
- PrimitivePublisher ::= <EventType>*
- CompositeSelector ::= <PrimitiveGummyModule> | <CompositeGummyModule>
- CompositeReactor ::= <PrimitiveGummyModule> | <CompositeGummyModule>
- CompositePublisher ::= <PrimitiveGummyModule> | <CompositeGummyModule>
- SharedVariable ::= <Identifier><Type>
- Variable ::= an acceptable variable definition in the language
- Method ::= an acceptable method definition in the language
- Class ::= an acceptable class definition in the language
- Identifier ::= an acceptable identifier in the language
- Type ::= $t \in T$ where T is a set of acceptable types in the language

Figure 4: A description of gummy modules

their publishers are only visible within the enclosing composite gummy module. Likewise, the publishers of composite publishers are recursively exported as the publishers of the enclosing composite gummy module; their selectors are only visible within the enclosing composite gummy module.

To implement event-driven applications via gummy modules, each EPCs can modularly be represented via primitive gummy modules. A set of correlated EPCs, which form an event processing network, can also be grouped together as one composite gummy module.

6 The GummyJ Language

We propose the version 2.0 of the GummyJ language[3], which is an extension to Java supporting gummy modules with composite and primitive structure. The details of GummyJ are explained by the example in Section 2.

6.1 Defining Events

The GummyJ language considers events as typed entities. An event type is a data structure defining a set of attributes, where an event is an instance that contains the values of attributes. There can be inheritance relation among event types. Listing 5 shows an excerpt of the specified event types. `EventType` is the predefined type, which is also super type of all event types. It defines the attributes `publisherID`, `targetID`, and `stacktrace`. These attributes are to keep the unique identifier of the publisher, the target of events, and the information about the active stack frames at the time events are published, respectively.

[3]https://github.com/malakuti/CompositeGummyModules

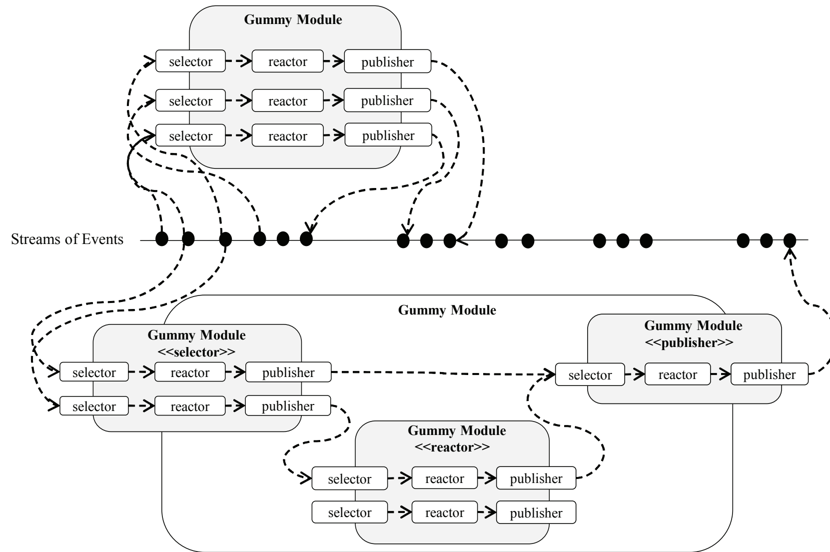

Figure 5: A schematic representation of gummy modules

New application-specific event types can be defined. For our example, we define the event type `FileEvent` as the super type of file-related events. This type defines the attributes `fileID` and `userrole` to keep information about the file and the role of the user that accesses the file, respectively. The event types `OpenEvent`, `ReadEvent`, `WriteEvent` and `CloseEvent` are specializations of `FileEvent`. The event types `TimeoutEvent` and `OutofOrderEvent` are defined to inform erroneous accesses to a file. Necessary information about the file and user, as well as amount of the time a file is kept open are maintained in the attributes of these event types. It is worth mentioning that in current implementation of GummyJ, attributes can be of the types `String`, `Long`.

```
1  eventtype EventType{String publisherID; String targetID;
2                     gummy.types.StackTrace stacktrace;}
3  eventtype FileEvent extends EventType{String fileID; String userrole;}
4  eventtype OpenEvent extends FileEvent{String mode;}
5  eventtype ReadEvent extends FileEvent{String content;}
6  eventtype WriteEvent extends FileEvent{String content;}
7  eventtype CloseEvent extends FileEvent{}
8  eventtype TimeoutEvent extends EventType{
9                  String fileID; String userID;Long opentime;}
10 eventtype OutofOrderEvent extends EventType{String fileID; String userID;}
```

Listing 5: Specification of event types

6.2 Defining Gummy Modules

There are various ways to program and modularize EPCs using gummy modules. We explain one possible case in the following, where we define each EPC as a separate gummy module to maximize reuse, and group them via the composite gummy module *PremiumUsersEPCs*.

Listing 6 define the primitive gummy module `FilterUsers`. Lines 3–4 define the shared variables `counter`, `userID` and `fileID`. Line 6 is the specification of a primitive publisher; i.e. the set of events published by the event method defined as a comma-separated list of event types. In this example, the method publishes events whose type polymorphically matches `FileEvent`.

```
1  gummymodule FilterUsers{
2  //the specification of shared variables
3    shared int counter;
4    shared String userID, fileID;
5
6    {FileEvent} //the specification of a primitive publisher
7    filter //the name of event method
8    { // the specification of a primitive selector
9      input in [OpenEvent, CloseEvent, ReadEvent, WriteEvent] &&
10     input.get("role") =="premium" &&
11     input.get("publisherID") == userID &&
12     input.get("fileID") == fileID &&
13     CounterHelper.check(counter, userID,fileID) < 5
14   }
15   {
16     //the specification of a primitive reactor
17     GummyJ.publish(input);
18   }
19 }
```

Listing 6: Modularizing the *FilterUsers* EPC with gummy modules

Line 7 defines the method name, and lines 8–14 define a primitive selector; i.e. the set of events filtered by the method. Here, we select events whose type matches `OpenEvent`, `CloseEvent`, `ReadEvent` or `WriteEvent`, and are published by the premium user `userID` to access the file `fileID`. The check in line 13 is to prevent the requests from a user who has kept the file open for more than five times in a day. The event selection expression refers to the Java helper class `CounterHelper`, which implements the functionality to check and reset the counter for the file-related events that are issued in a day. Lines 15–18 define a primitive reactor; i.e. a program in Java. Here, we only publish the selected event, if any.

In GummyJ, the keyword `input` refers to the current event being processed by the gummy module; naturally the event represented via `input` is a subtype of `EventType`, which is the predefined super type for all event types. It is possible to access event attributes via the method `get()`. If the specified attribute is not

defined in the event type, an exception will be thrown. The type of the events published from within event methods must polymorphically match at least one of the types specified in the signature of the method, otherwise an exception will be raised.

Lines 1–29 of Listing 7 defines the primitive gummy module `CheckTimeout`. Lines 3–5 define the local variables `opened`, `timer`, and `processed`, which represent whether the file is already opened, the amount of time that the file is kept open, and the last file-related event that has been processed, respectively.

```
1  gummymodule CheckTimeout{
2   //the specification of local variables
3    Boolean opened;
4    Long timer;
5    Long T = new Long(10);
6
7   //the specification of local methods
8    Long init (){...}
9    Long update (){...}
10   Long reset (){...}
11
12   //the specification of event methods
13   {TimeoutEvent} checkOpenEvent {input in [OpenEvent]} {
14     if (! opened) {
15       opened = true;
16       timer = init();
17       while (is_opened() && timer < T) timer = update();
18       if (timer >= T && is_opened()) {
19         TimeoutEvent error = new TimeoutEvent();
20         output.fileID = input.get("fileID");
21         output.userID = input.get("publisherID");
22         output.opentime = timer;
23         GummyJ.publish(error);
24       }
25     }
26     }
27    {} checkReadWriteEvents {input in [ReadEvent, WriteEvent]} {timer = reset();}
28    {} checkCloseEvent {input in [CloseEvent]} {timer = reset(); opened = false;}
29 }
30 gummymodule CheckOrder{
31     {OutofOrderEvent} check {input in [FileEvent]}
32     {
33         // check order implementation
34         ...
35       OutofOrderEvent error = OutofOrderEvent();
36       GummyJ.publish(error);
37     }
38 }
```

Listing 7: Modularizing the *CheckTimeout* and *CheckOrder* EPCs with gummy modules

Lines 8–10 define a set of local methods to manipulate the timer. Lines 13–26 define the event method `checkOpenEvent`, which selects events whose type polymorphically matches `OpenEvent`, and publishes events of the type `TimeoutEvent`. If a file is left open for an excessive amount of time, lines 19–23 create an instance of the event type `TimeoutEvent`, initialize its attributes, and publish it to the runtime environment of the GummyJ language. Line 27 defines the event method `checkReadWriteEvents`, which resets the timer when a read or write request is received for the file. Likewise line 28 processes the events of the type `CloseEvent`, resets the timer and the variable `opened`.

Lines 30–38 define the primitive gummy module `CheckOrder` to check the order in which file events are issued.

Listing 8 defines the primitive gummy module `FilterEvents`, which selects events of the type `TimeoutEvent`, increases the counter, and publishes the event further if the counter exceeds the value 5.

```
1 gummymodule FilterEvents{
2     shared int counter;
3   {TimeoutEvent} filter {input in [TimeoutEvent]}{
4     counter ++;
5     if (CounterHelper.check(counter, input.get("userID"), input.get("fileID"))>=5)
6       GummyJ.publish(input);
7   }
8 }
```

Listing 8: Modularizing the *FilterEvents* EPC with gummy modules

Listing 9 defines the composite module `PremiumUsersEPCs`, whose constructor receives a user ID and file ID as its parameters to distinguish the instances of this gummy module for each user and file. The gummy module defines a shared variable named as `counter` to maintain the number of times a user has kept a file open for an excessive amount of time. In addition it defines two shared variables *userID* and *fileID*.

Line 10 defines the selector part of this gummy module, which is an instance of the gummy module `FilterUsers` represented via the variable `userselection`. Line 11 defines the first reactor of the composite gummy module, which is an instance of the gummy module `CheckOrder`. The second reactor is defined in 12, which is an instance of the gummy module `CheckTimeout`. This indicates that first the order of events and then the timeout case are checked. Line 13 defines the first publisher of `PremiumUsersEPCs`, which is an event of the type `OutofOrderEvent`. Line 14 defines the second publisher, which is an instance of the gummy module `FilterEvents`.

Using this structure, the events that are published to the application are visible to `userselection`, which filters the events of interest and forwards them inside the composite gummy module. These events are visible to the reactors and publishers of this module, but are only of interest for the reactors. The events that are published by the reactors are processed by the publishers of the composite gummy module.

```
1  gummymodule PremiumUsersEPCs{
2      //the specification of shared variables
3      shared int counter; shared String userID, fileID;
4
5      //the specification of constructor
6      public PremiumUsersEPCs (String UID, String FID){
7        userID = UID; fileID = FID;
8      }
9
10   selector FilterUsers userselection = new FilterUsers();
11   reactor CheckOrder orderchecker = new CheckOrder();
12   reactor CheckTimeOut timeoutchecker = new CheckTimeOut();
13   publisher OutofOrderEvent ordererror = new OutofOrderEvent();
14   publisher FilterEvents timeouterror = new FilterEvents();
15 }
```

Listing 9: Modularizing the event processing network with gummy modules

Gummy modules that are enclosed within a composite gummy module (e.g. `PremiumUsersEPCs`) share the same runtime identifier as the composite gummy module, and can access the shared variables defined within the composite gummy module. For example, since `PremiumUsersEPCs` encloses `FilterUsers` and `FilterEvents`, their shared variable `counter` refers to the same memory location as the variable `counter` in `PremiumUsersEPCs`.

In most modern object-oriented languages, when the return type of a method is not void, the compiler raises an error if an execution path in the method does not return a value/exception to its caller. This is to comply with the request-reply communication, where a request must be replied by providing a return value to the caller. In the event-based communication, events are broadcast, and if they are of the interest to any recipient, they will be processed. This means that event producers do not expect a reply, and some events may not be processed at all. Therefore, to reduce the amount of events that are of no interest to any recipient, the GummyJ compiler does not force all the execution paths inside event methods to publish an event. This is shown for example in the `checkOpenEvent` event method in Listing 7, where only timeout events are published when needed.

6.3 Integrating with Objects/Aspects

Gummy modules can explicitly be instantiated; when a composite gummy module is instantiated, all of its enclosed gummy modules are instantiated too. Assume for example that we would like to apply the module `PremiumUsersEPCs` to the users with the identifier *U1–U4* who access the files with the identifier *F1.txt–F4.txt*, respectively. GummyJ maintains a pool of gummy module instances; each instance is designated via its unique index in this pool. To reduce the amount of necessary bookkeeping code, in the spirit of Ptolemy and Esper, GummyJ offers an API to instantiate gummy modules, and to register them in

the runtime environment of the language. An example usage of this API is in lines 3–6 of Listing 10 shows. One can also destroy the instances if needed, e.g. lines 8–9.

Event types are translated to Java classes whose instances are used to represent events. As shown in line 19 of Listing 10, GummyJ offers an API, which can be used by pure objects, aspects and even gummy modules to publish events in an asynchronous way.

As shown in lines 22–27 of Listing 10, legacy Java objects in applications can also receive events from gummy modules. To this aim, they must define methods that receive desired event types as their argument, and must register in GummyJ to receive events from a specific module instance referenced by its unique index.

```
1  public class Usage {
2    public void manage (){
3     PremiumUsersEPCs epn_1 =
4               GummyJ.instantiate(PremiumUsersEPCs,"U1","F1.txt");
5     PremiumUsersEPCs epn_2 =
6               GummyJ.instantiate(PremiumUsersEPCs,"U2","F2.txt");
7    ...
8    GummyJ.destroy(epn_1);
9    GummyJ.destroy(epn_2);
10   }
11 }
12 public class Producer {
13   public void produce (){
14   OpenEvent open = new OpenEvent();
15   open.fileID = "...";
16   open.userrole = "premium";
17   open.publisherID = "U1";
18   ...
19   GummyJ.publish(open);
20   }
21 }
22 public class Consumer {
23   public Consumer (){
24     GummyJ.register (this, "consume", epn_1);
25   }
26   public void consume (TimeoutEvent event){...}
27 }
```

Listing 10: An example usage of gummy modules

The runtime event processing semantics of gummy modules is explained in the next section. As an example, assume that the module `PremiumUsersEPCs` is instantiated, and the event `open` is published to the runtime environment of GummyJ. The event is visible to all instantiated gummy modules, and is evaluated against the selectors of the gummy modules. The event matches

the selector of `epn_1`, which forwards it inside `epn_1`. The event is visible to the reactors and publisher of `epn_1`, but is only of interest for its reactors. The reactors process the event concurrently. Since this event does not violate the expected occurrence order of the events, the reactor `timeout` restarts the timeout timer. If other file events are not issued within the expected period, `timeout` produces an event of the type `TimeoutEvent`, which is received by the publisher `filterevents` of `epn_1`.

7 Event Processing Semantics of GummyJ

An application developed in GummyJ consists of a set of gummy modules, environmental objects and application objects that produce and/or consume events. Due to the support for asynchronous event publishing, event producers and gummy modules are executed concurrently.

Figure 6 provides an abstract view of the gummy modules at runtime. The runtime manager of GummyJ has a global event queue, in which the events published to the runtime manager are maintained, and are served in an FIFO manner. The events in the global event queue are visible to the selectors of gummy modules; the publishers of the modules may insert events in this queue.

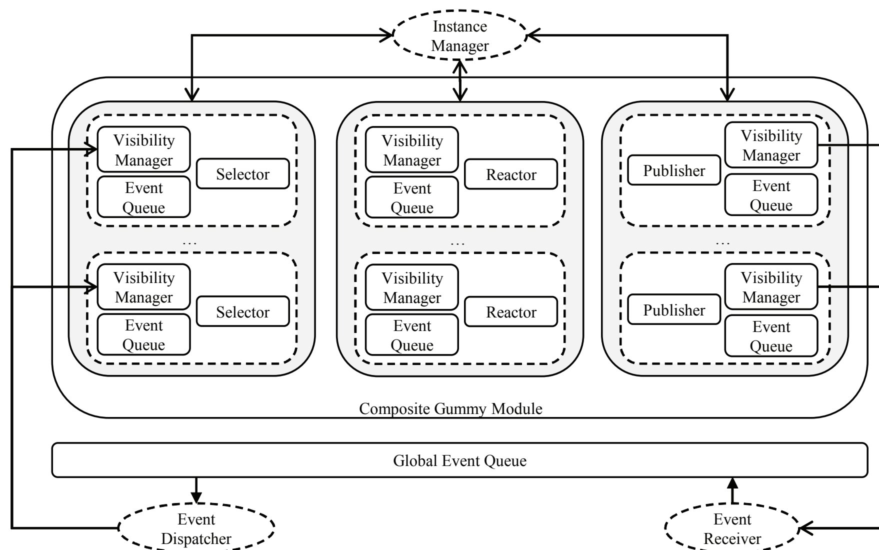

Figure 6: The runtime representation of gummy modules

The *Instance Manager* object maintains the meta-data about gummy modules and their internal structure. It also handles the requests to construct and destruct instances of gummy modules. The *Event Dispatcher* object dispatches global events to gummy module instances. The *Event Receiver* object gets events and inserts them into the global event queue.

The event methods in primitive gummy module, and the selectors, reactors and publishers in composite gummy modules process events concurrently. Therefore, each have their own local queue to maintain the events that are visible to them for processing. Each selector, reactor and publisher is associated with a *Visibility Manager* object, which receives visible events and inserts them into the relevant local event queue. For example, the *Visibility Manager* of selectors receives the events that are visible to selectors (i.e. events in the global event queue), and inserts them in the local event queue of the selectors.

Likewise, each reactor is associated with a *Visibility Manager* object, which receives the events published by selectors, and inserts them in the corresponding local event queue. The same applies to publishers except that their visible events are the ones published by selectors and/or reactors. This means that the event flow inside a gummy module forms a direct acyclic graph, from selectors to reactors and publishers.

All these objects are executed concurrently via independent Java threads. The concurrent accesses to event queues and shared variables in gummy modules are synchronized via Java synchronization mechanisms.

As for other general-purpose abstractions, the ways that gummy modules are defined, communicate with each other and with their environment depends on application requirements. Nevertheless, to get an insight about the behavior of applications developed in GummyJ, we adopt the UPPAAL toolbox [2] to simulate possible behavior of gummy modules independently from any specific application.

The choice of UPPAAL is justified by the fact that gummy modules are executed concurrently to each other and to the event producers in their environment; besides, the internal elements of composite gummy modules are executed concurrently to each other. UPPAAL facilitates modular modelling of software behavior using separate automata, which are executed concurrently and are composed together to represent the final behavior of the software.

In UPPAAL, each automata may have a set of local variables and functions, and may be instantiated multiple times. Automata communicate via shared variables and channel expressions such as *c!* and *c?*. The channel expression *c!* in an automaton is comparable to an asynchronous method invocation; this invocation is received by the expression *c?* in another automaton.

The abstract behavior of the *Instance Manager* entity is shown in Figure 7. This automaton receives an instantiation request via *instantiate?*, and constructs an instance of the gummy module by invoking the function *instantiate_gummymodule()*. Afterwards, via the channel *activate!*, it broadcasts a request to activate the selectors, reactors and publishers of the module and their *Visibility Manager* objects.

The request to destroy an instance of a gummy module is received via the channel *destroy?*. Since the gummy module may be currently processing an event, the destruction is delayed until the selectors, reactors and publishers of the module finish processing their current event. To implement this, *Instance Manager* first marks the gummy module to be destroyed via invoking the function *mark_to_destroy()*, and then waits in the location *Destroying* until

the selectors, reactors and publishers of the gummy module are ready to be destroyed. Afterwards, it destroys the instance of the gummy module via invoking the function *destroy_gummymodule()*.

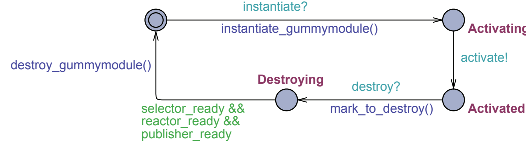

Figure 7: Abstract behavior of *Instance Manager*

Figure 8 shows the abstract behavior of *Event Receiver*, which receives input events via channel *event_to_global?*, and inserts them in the global event queue. The abstract behavior of *Event Dispatcher* is shown in Figure 9, which checks whether there is any active gummy module. If not, it just discards the events that are inserted in the global event queue via invoking *dequeue()*. Otherwise and if there is an event in the global event queue, it notifies the *Visibility Manager* of the selectors via the channel *event_from_global!*. It waits until *Visibility Manager* informs it that the event is received by *Visibility Manager*. This notification is done via the channel *ready?*. As a result, *Event Dispatcher* removes the event from the global event

It Afterwards, it waits until via the channel *ready?* it is informed by *Visibility Manager* that the event is received by *Visibility Manager*. *Event Dispatcher* consequently removes the event from the global event queue via invoking *dequeue()*.

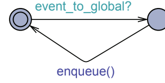

Figure 8: Abstract behavior of *Event Receiver*

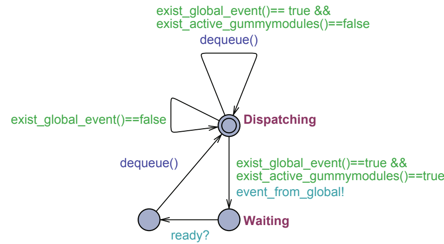

Figure 9: Abstract behavior of *Event Dispatcher*

Figures 10 shows the abstract behavior of the *Visibility Manager* objects for selectors. When an instance of a gummy module is constructed by *Instance*

Manager, the automaton in Figure 10 synchronizes with *Visibility Manager* via the channel *activate?*, and as a result the instance of *Visibility Manager* are activated to receive events.

If there is a new event in the global event queue, *Visibility Manager* is informed via the channel *event_from_global?*. As a result, it retrieves a copy of the event via the function *retrieve_selector_visible_event()*, inserts it in the local queue of the selector, and via the channel *ready!* it informs *Event Dispatcher* that it is ready to receive the next event from the global event queue, if any. It then takes a transition to the location *Activated*. If the gummy module is marked to be destroyed, *Visibility Manager* stops receiving events from the global queue by taking a transition to the initial state.

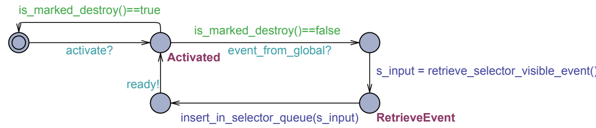

Figure 10: Abstract behavior of *Visibility Manager*

Figure 11 shows the abstract behavior of selectors, which are executed concurrently to *Instance Manager* and *Visibility Manager*. After a selector is activated, it starts processing the events that exist in its local queue. It retrieves the event at the front of the queue via the function *get_next_event()*, and in the location *EventMatching* it checks whether the event is of the interest for the selector. If so, a transition is taken to the location *EventProcessing*.

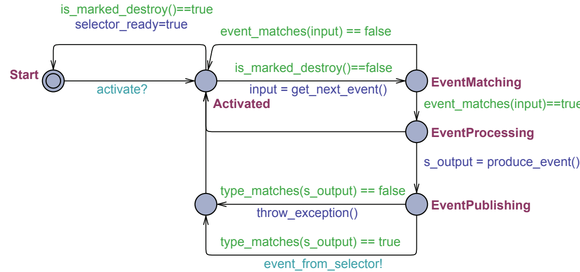

Figure 11: Abstract behavior of selectors

Depending on the application logic, various operations can be performed in this location. For example, the selector may do nothing; this is shown via a transition to the location *Activated*. It may also produce a new event via the function *produce_event()*, and try to publish it. In this case, the function *type_matches()* checks whether the event type polymorphically matches the publishers type for the selector. If not, an exception is raised; otherwise, the reactors and publishers of the composite gummy module are informed of the event via the channel *event_from_selector!*. This will be handled by the *Visibility Manager* objects of

reactors and publishers, which will receive a copy of the produced event and will insert it in the local event queues of the reactors and publishers. The event processing semantics of reactors and publishers are similar to selectors, except that access different event queues.

After the selector finishes processing the current event, a transition is taken to the location *Activated* in Figure 11. Here, it is checked whether the composite gummy module is marked to be destroyed. If so, a transition is taken to the initial state and the remaining events in the local queue are discarded.

8 Discussions

As shown in Listing 5, new types of events can be defined, and can explicitly be published to GummyJ via its API. Alternatively, we could adopt the same approach as most AO languages, where the set of supported events (join points) is defined by the join point model of the language. The advantage of these languages is that the occurrences of events (join points) are automatically designated in programs by their compiler. However, it has been well-studied that the join point model of AO languages is not rich enough to cover all desired events and event attributes. Consequently, there are various proposals to support extensible join point models, in which programmers have to explicitly annotate desired code blocks in programs as join points [10, 34]. Therefore, we believe that a language must support explicit event definition and announcement; otherwise, its applicability for different application domains may reduce.

As shown Listings 8, 7, 6, for example via the gummy module `CheckTimeout`, individual EPCs can be modularized via primitive gummy modules. The signature of event methods specify the set of events that are processed and published by the methods.

as Listing 9 shows, a set of correlated EPCs can be reused to define an event processing network. The interfaces of composite gummy modules (i.e. selectors and publishers) are separated from its implementation (i.e. reactor). Such a composite structure facilitates separating event selection, event processing and event publishing logic from each other, and yet composing them together under one module abstraction.

The visibility of events is controlled via composite gummy modules. This way of defining scopes complies with the concept of modularization, which emphasizes on hiding certain aspects of modules from being accessible to its external environment.

Gummy modules communicate with each other and with the event producers and consumers in an asynchronous way. Therefore, there is no synchronization coupling among them. There is no explicit reference among gummy modules and event producers/consumers, implying that there is no space coupling among them. Event producers publish events to the GummyJ runtime; if there is any gummy module registered, the events are provided them for processing. Event producers and gummy modules can have different lifetime. Therefore, there is no time coupling among event producers and gummy modules.

9 Related Work

In Section 4, we evaluated a large set of languages that support event-driven programming. We will now evaluate other categories of relevant research.

Various complex event processing middleware have been proposed in the literature [7, 3], which provide means to process large amount of events in an efficient way. Such middleware helps to separate event processing from the core functionality of applications. However, such a separation may not be desirable when event processing is a part of applications logic and must access application-specific information. Nevertheless, such middlware are useful in the large-scale applications that require fast processing of a massive amount of events in short time. Therefore, we would like to provide means to integrate GummyJ with such middlware, to benefit from them for initial processing of events, such as filtering out irrelevant events or transforming the format of events.

Rule-based systems consist of a set of rules defined as conditional expressions, a set of facts, and an engine that controls the activation and application of the rules [22]. Active rules [11], which are also known as event-condition-action (ECA) rules, are an example of rule-based database systems. When an event occurs, conditions are evaluated, and if they are satisfied, an action is triggered. In [6], the concept of event-based stratification has been introduced, which facilitates separating and classifying rules based on input events that trigger the rules, and the output events that are produced by the rules. The authors focus on criteria for stratifying rules such that the stratified rules guarantee the termination and confluence properties. Unlike this approach, our focus is on applications, and we proposed a module abstraction and a language to facilitate modularizing event-driven applications. Adopting the concept of gummy modules at the database level for modularizing rule sets is an interesting direction of research for future.

The concept of *scopes* is proposed to limit the visibility of events between producers and consumers [18]. A scope can have interfaces, which define the set of events that can enter or leave the scope. Scopes can also have a composite structure. An SQL-like language is provided to define scopes and to add/remove components to the scopes. An implementation of scopes is also implemented in the REBECA framework in Java, which offers dedicated APIs to attach components to scopes. The first distinguishing characteristic of our approach is that we proposed a new kind of module abstraction, its implementation in a programming language, and its integration with other modules. In contrast to scopes whose sole purpose is to limit the visibility of events, composite gummy modules are reusable units of computations enclosing smaller units of computations. Unlike scopes, the selectors, reactors and publishers of a composite gummy modules are also defined in terms of gummy modules. This uniformity increases the compositionality of applications, facilitates defining course-grained modules in a flexible manner, and enables defining complex event filtering semantics.

There are several proposals for event-based coordination of concurrent processes [8, 21]. There are also proposals in functional-reactive programming [33, 13], which provide dedicated abstractions to model time-changing values usually

termed as signals. Where signals and events are analogous, event processing languages mainly focus on defining composite events from patterns of primitive ones, involving content-based and temporal constraints on the patterns. Therefore, event processing languages and reactive languages can be regarded complementary [29].

The actor model [4] is a widely-adopted model for implementing concurrent computation. This model is implemented in various object-oriented and functional programming languages. The current implementations [5] of this model support a limited form of message selection and publishing, with the same shortcomings as the languages evaluated in Section 4. Although concurrency is one characteristic of gummy modules, we have a different focus than actors; our main focus is on providing suitable abstractions for modularizing event-driven applications. We believe that gummy modules provide a strong base to implement complex message passing semantics of actors in a modular way. We would like to explore this in future.

In our previous work [25], we have proposed *emergent gummy modules* as event-based modules that encapsulate their instantiation and destruction semantics. Emergent gummy modules are useful in representing EPCs that have transient nature, for example, the ones representing the appearance or disappearance of a certain behavior in the environment. As explained in Section 4, emergent gummy modules and their implementation in the first version of GummyJ do not fulfill our requirements, because they do not have a composite structure and do not specify the events that they publish. Our proposal in this paper solves these shortcomings. In this report, we did not focus on the emergent characteristics of modules; modules must be explicitly instantiated or destroyed from within Java objects as shown in Listing 10. Therefore, the proposed module abstraction can be used for programming EPCs that do not have transient nature. Supporting composite emergent gummy modules is the focus of our future work.

10 A Close Look at Event-based Modularization

As software applications are finding their way in different domains, new kinds of concerns may appear in such applications. The so-called event processing concerns are the key concerns in event-driven applications, which implement the functionality to process events.

During past decades, modularization mechanisms have evolved to facilitate modularizing different kinds of concerns that appear in applications. We briefly summarize the evolution of four modularization mechanisms in Figure 12.

In procedural modularization, a program is divided into a set of procedures/-functions, which invoke each other with zero or more call arguments. The callee procedure/- function executes the call, and it may invoke on other procedures. This paradigm eventually evolved to the object-oriented modularization to reduce the complexity and to enhance the flexibility and the evolvability of software systems.

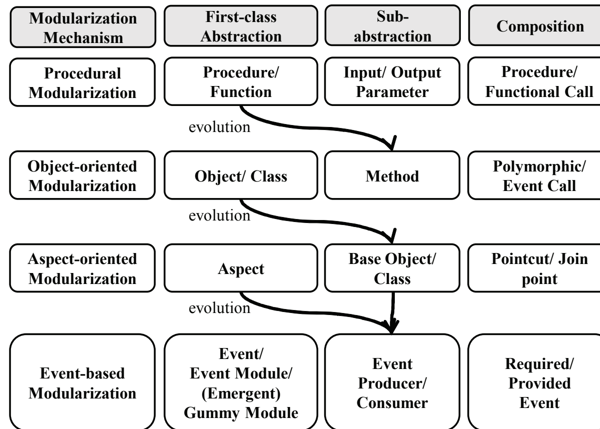

Figure 12: An evolution of modularization mechanisms

Objects/classes are means to structure software systems; objects are considered a better match for the abstractions of the real world. An object groups a set of attributes and procedures (methods) together. The methods can be explicitly invoked, or if the language offer an event-delegate mechanism, they can be invoked implicitly via event announcement. Objects are useful in representing hierarchically-structured entities of the real world. However, they fall short in representing crosscutting concerns, which do not fit into such hierarchical structures. Consequently, the implementation of such concerns scatters across and tangles with objects.

Aspect-oriented modularization has been introduced as a solution. Here, aspects are introduced as dedicated abstractions to modularize crosscutting concerns. Objects are treated as the base to which aspects apply via pointcuts and join points mechanisms.

The current trend in language support for event-driven programming is to adopt an existing procedural, object-oriented or aspect-oriented language and extend it with some event processing features. As we discussed in this report, such extensions are rather ad-hoc, and lead to degradation of modularity and decoupling in various ways.

We have been developing the concept of *event-based modularization* [26] as a dedicated modularization mechanism for event-driven applications. We consider the computation as one or more sequences of events, which may be produced by various entities. A module is defined as a group of correlated events and the reactions to them. We have introduced *event modules* [27], *emergent gummy modules* [25] and *gummy modules* as three dedicated module abstractions. The shortcomings of event modules and emergent gummy modules are explained in Section 4. In this report, we focused on gummy modules, and their advantages for modularizing event-driven applications.

As Figure 12 shows, event-based modularization can be regarded as the successor of current modularization mechanisms for event-driven applications. Events and gummy modules are first-class abstractions, and modules are composed with each other via publishing and consuming events. Conventional objects and aspects may become event producer and/or consumer too.

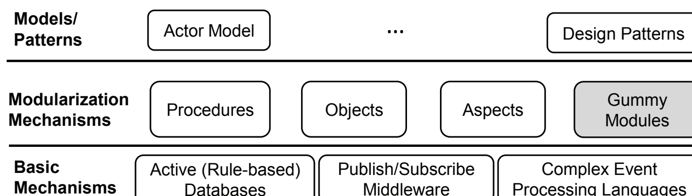

Figure 13: The position of gummy modules

Figure 13 depicts the position of gummy modules in the state of the art. Gummy modules are orthogonal to other module abstractions, dedicated for event-driven programming. They may benefit from the services that are provided by active database, publish/subscribe middleware and complex event processing engines for initial processing of events. Gummy modules can be adopted to implement various communication models such as the actor model as well as design patterns.

In this report, we focused on adopting event-based modularization at the programming language level. Nevertheless, this concept can also be applied to other stages of software development process. In [28], we showed the use of event modules at the architectural level to modularly compose multiple applications.

11 Conclusions and Future Work

We discussed that current module abstractions and their implementation in imperative languages have significant shortcomings to properly modularize event processing concerns. We discussed gummy modules as novel module abstractions, which have well-defined event-based interfaces. Gummy modules can have a composite structure, enabling complex event selection, processing and publishing semantics be modularized and integrated together as one composite gummy module. Composite gummy modules are also means to define visibility scopes of events.

As Figure 2 shows, GummyJ 2.0 is in the category of languages supporting event quantification. We envision to extend GummyJ such that it can also support stream/complex event processing, as well as publish/subscribe paradigm. The former will be achieved by offering languages to reason about correlations of events, and the latter will be achieved by supporting distribution in gummy modules. Both of these language categories emphasize processing large amounts of events in an efficient way. Therefore, we would also like to study means to improve the performance of GummyJ in processing large amount of events. Last

but not least, we will also study advanced composition mechanisms such as inheritance in gummy modules.

Acknowledgement

The author is supported by the German Research Foundation (DFG) in the Collaborative Research Center 912 "Highly Adaptive Energy-Efficient Computing". The author would like to thank Mehmet Aksit and Uwe Assmann for their feedback on early versions of this work.

References

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

[2] UPPAAL. http://www.uppaal.org/.

[3] Asaf Adi. IBM Active Middleware Technology Overview. Technical report, IBM Haifa Labs, 2006.

[4] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[5] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2013.

[6] Elena Baralis, Stefano Ceri, and Stefano Paraboschi. Modularization Techniques for Active Rules Design. *ACM Trans. Database Syst.*, 21(1):1–29, March 1996.

[7] Raphaël Barazzutti, Pascal Felber, Christof Fetzer, Emanuel Onica, Jean-françois Pineau, Marcelo Pasin, Etienne Rivière, and Stefan Weigert. StreamHub : A Massively Parallel Architecture for High-Performance Content-Based Publish / Subscribe Categories and Subject Descriptors. In *Proceedings of DEBS*. ACM, 2013.

[8] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern Concurrency Abstractions for C#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, 2004.

[9] Juergen Boldt. The Common Object Request Broker: Architecture and Specification. Technical report, Object Management Group, July 1995.

[10] Walter Cazzola and Edoardo Vacchi. Fine-grained Annotations for Pointcuts with a Finer Granularity. In *SAC '13*. ACM, 2013.

[11] Umeshwar Dayal. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann Publishers Inc., 1994.

[12] Arjan de Roo, Hasan Sozer, and Mehmet Aksit. Composing Domain-specific Physical models with General-purpose Software Modules in Embedded Control Software. *Software Systems Modeling*, 13(1):55–81, 2014.

[13] Conal M. Elliott. Push-pull Functional Reactive Programming. In *Haskell '09*. ACM, 2009.

[14] Opher Etzion and Peter Niblett. *Event Processing in Action*. Manning, 2010.

[15] Patrick Eugster. Type-based Publish/Subscribe: Concepts and Experiences. *ACM Trans. Program. Lang. Syst.*, 29(1), 2007.

[16] Patrick Eugster and K. R. Jayaram. EventJava: An Extension of Java for Event Correlation. In *Proceedings of ECOOP '09*. Springer-Verlag, 2009.

[17] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The Many Faces of Publish/Subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[18] Ludger Fiege, Gero Mühl, and Felix C. Gärtner. Modular Event-based Systems. *Knowl. Eng. Rev.*, 17(4):359–388, 2002.

[19] Vaidas Gasiunas, Lucas Satabin, Mira Mezini, Angel Núñez, and Jacques Noyé. EScala: Modular Event-driven Object Interactions in Scala. In *Proceedings of AOSD '11*. ACM, 2011.

[20] M. Haahr, R. Meier, P. Nixon, V. Cahill, and E. Jul. Filtering and Scalability in the ECO Distributed Event Model. In *Software Engineering for Parallel and Distributed Systems, 2000. Proceedings. International Symposium on*, pages 83–95. IEEE, 2000.

[21] Philipp Haller and Tom Van Cutsem. Implementing Joins Using Extensible Pattern Matching. In *COORDINATION '08*. Springer-Verlag, 2008.

[22] Frederick Hayes-Roth. Rule-based Systems. *Commun. ACM*, 28(9):921–932, 1985.

[23] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. EventCJ: a Context-oriented Programming Language with Declarative Event-based Context Transition. In *Proceedings of AOSD '11*. ACM, 2011.

[24] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of ECOOP' 97*. Springer-Verlag, 1997.

[25] Somayeh Malakuti and Mehmet Aksit. Emergent Gummy Modules: Modular Representation of Emergent Behavior. In *Proceedings of GPCE '14*. ACM, 2014.

[26] Somayeh Malakuti and Mehmet Aksit. Event-based Modularization: How Emergent Behavioral Patterns Must Be Modularized? In *Proceedings of FOAL '14*. ACM, 2014.

[27] Somayeh Malakuti and Mehmet Aksit. Event Modules - Modularizing Domain-Specific Crosscutting RV Concerns. *T. Aspect-Oriented Software Development*, pages 27–69, 2014.

[28] Somayeh Malakuti and Mariam Zia. Adopting Architectural Event Modules for Modular Coordination of Multiple Applications. Technical report, Technical University of Dresden, 2015.

[29] Alessandro Margara and Guido Salvaneschi. Ways to React : Comparing Reactive Languages and Complex Event Processing. In *Reactivity, Events and Modularity workshop*, 2013.

[30] Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Roşu. An Overview of the MOP Runtime Verification Framework. *International Journal on Software Techniques for Technology Transfer*, pages 249–289, 2011.

[31] Microsoft Corporation. C# language specification. http://msdn.microsoft.com/en-us/vcsharp/aa336809.aspx.

[32] Hridesh Rajan and Gary Leavens. Ptolemy: A Language with Quantified, Typed Events. In *Proceedings of ECOOP '08*, LNCS. 2008.

[33] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications. In *Proceedings of MODULARITY '14*. ACM, 2014.

[34] Friedrich Steimann, Thomas Pawlitzki, Sven Apel, and Christian Kästner. Types and Modularity for Implicit Invocation with Implicit Announcement. *ACM Trans. Softw. Eng. Methodol.*, 20:1:1–1:43, 2010.

[35] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 179–196. Springer-Verlag, 2002.