# Specification of the GummyModule Language

Somayeh Malakuti

Software Engineering Group,
University of Twente, The Netherlands
s.malakuti@ewi.utwente.nl
Version 1.0
December 2012

**Abstract.** The GummyModule language is an extension to the Java language and is the successor of the EventReactor language introduced in [1, 2]. The GummyModule language adopts the linguistic constructs of offered by EventReactor to define event types, events and to publish events. As for EventReactor, GummyModule facilitates defining the functionality of modules via domain-specific languages (DSL), and is open-ended with new DSLs. This technical report first explains the syntax of the GummyModule language. Afterwards, it discusses the algorithm adopted by the runtime environment of the GummyModule language to process events.

## 1 Syntax of the GummyModule Language

### 1.1 Specification of Event Types

In GummyModule, the inter-module interactions is performed using events, which are typed entities. Listing 1 shows the structure of the specification of event types that can be used by programmers to define new event types in GummyModule.

```
1  eventtype ::= 'eventtype' <name> 'extends' <super−name> '{'
2              ('staticcontext' ':'
3                <attr−name> ':' <Java type> ';'
4                ...
5              )?
6              ('dynamiccontext' ':'
7                <attr−name> ':' <Java type> ';'
8                ...
9              )?
10            '}'
```

**Listing 1.** The specification of event types

As the identifier `<name>` in line 1 shows, an event type is identified by a name that must be unique in GummyModule. As the clause `'extends' <super-name>` shows, an event type extends another event type. Lines 2 to 9 show that each

new event type may define new attributes in the parts `staticcontext` and/or `dynamiccontext`. Each attribute is defined via a name and a type, which can be any valid type in the Java language.

GummyModule provides five built-in data structures `EventType`, `Activation`, `Deactivation`, `BaseEvent`, and `MethodBased` . As Listing 2 shows, the data structure `EventType`, which is the super[1] data structure for all event types, defines the attribute `type` in its static context, and defines the attributes `publisher`, `thread`, `timestamp`, `stacktrace`, and `returnflow` in its dynamic context. The attribute `type` maintains a reference to the event type; the attributes `publisher`, `thread` and `timestamp` are to maintain the unique identifier of the publisher, the execution thread in which an event is published and the timestamp of the event. The type `StackTrace` is defined by GummyModule to keep the list of active stack frames. The type `Flow` is defined by GummyModule as an enumeration with the fields `Continue`, `Exit` and `Return`. The field `Continue` means that the flow of execution must not be changed. The field `Exit` means that the execution of program must terminate. The field `Return` the flow of execution must return to the publisher. More details about controling the flow of execution using these fields at runtime is provided inčiteMalakuti2011.

The event types `Activation` and `Deactivation` are used to inform the runtime environment of GummyModule that an instance of a gummy module must be constructed and destructed, respectively.

```
1  eventtype EventType {
2    statcicontext:
3      type : Object;
4    dynamiccontext:
5      publisher : Object;
6      thread : long;
7      timestamp : Long;
8      stacktrace : StackTrace;
9      returnflow : Flow;
10 }
11 eventtype Activation extends EventType {}
12 eventtype Deactivation extends EventType {}
```

**Listing 2.** The specification of built-in super event types

The data structure `BaseEvent`, shown in Listing 3 is the super for base events, i.e. the events published by the base objects. While programmers can define new kinds of base events, GummyModule itself defines the event type `MethodBased`, which is internally used by gummy module to represent the following state changes in base software: a) before invocation of methods; b) after invocation of methods; c) after invocation and immediately before execution of methods; and d) after execution of methods, which have terminated normally. These events represent the join points that are supported by many of the current aspect-oriented languages [3–5]. Listing 3 shows the definition of `MethodBased`.

[1] Super in the object-oriented terminology.

Here, line 2 defines the event type `MethodBased` as an extension of the event type `BaseEvent`, because it represents the state changes in the base software. Line 4 defines the attribute `kind` of the type `MethodBasedEvents` that is an enumeration defined by gummy module with four fields: `BeforeInvocation`, `AfterInvocation`, `BeforeExecution` and `AfterExecution`.

```
1  eventtype BaseEvent extends EventType {}
2  eventtype MethodBased extends BaseEvent{
3    staticcontext:
4      kind : MethodBasedEvents;
5      signature : Signature;
6      module : Module;
7      location : CodeSegment;
8
9    dynamiccontext:
10     args : Object [];
11     target : Object;
12 }
```

**Listing 3.** The specification of MethodBased event type

Lines 5 to 7 define the attributes `signature`, `module` and `location` of the types `Signature`, `Module` and `CodeSegment`, respectively. These types are defined by gummy module. The attribute `signature` is used to keep the signature of the method whose invocation and/or execution will be represented as an event. The attribute `module` is used to keep a reference to the module in which the corresponding method is defined. The attribute `location` is used to keep information about the location at program code where the event must be triggered.

The attribute `args` defined in the part `dynamiccontext` is used to keep the arguments of the corresponding method invocation and/or execution. The attribute `target` is used to keep a reference to the receiver of a method invocation.

The event type `MethodBased` inherits the attributes `publisher`, `thread`, `stacktrace` and `returnflow` from the event type `BaseEvent`. If for the attribute `kind` the value `BeforeInvocation` or `AfterInvocation` is, the module that performs the invocation is considered as the publisher; otherwise, the module that performs the execution is considered as the publisher. In any case, the attribute `thread` keeps the unique identifier of the thread of execution in which the method invocation and/or execution is performed, and the attribute `stacktrace` keeps the list of the active stack frames.

As for many current aspect-oriented languages [3–5] that statically identify the join points in base software, the compiler of GummyModule can extract the standard events from base software, and define them in GummyModule. If needed, the automatic definition of standard events by the compiler can be disabled by programmers so that they can limit the events that are defined in GummyModule.

### 1.2 Specification of Events

Listing 4 shows the structure of the specification of events, which can be used by programmers to define new events in GummyModule. The identifiers `<name>` and `<eventtype-name>` in line 1 indicate that each event is identified by a name that must be unique in GummyModule, and has a type. Lines 2 to 5 indicate that the attributes defined in the part `staticcontext` of the corresponding event type must be initialized in the definition of an event, if any.

```
1 event::= 'event' <name> 'instanceof' <eventtype−name> '{'
2         ('staticcontext' ':'
3            <attr−name> '=' <value> ';'
4            ...
5         )?
6       '}'
```

**Listing 4.** The specification of events

### 1.3 Publishing Events

In GummyModule, each event specified as Listing 4 is translated to a Java object that defines the specified attributes. To publish an event, the corresponding Java object must be instantiated, the attributes specified in the part `dynamiccontext` must be initialized, and finally the instantiated object must be sent to the runtime environment of gummy module using the provided API by gummy module. An example is shown in [2].

The compiler of GummyModule modifies the base software to publish the standard events. As for the definition of standard events, this feature can be disabled so that programmers can write desired publishing code.

### 1.4 Specification of Domain-Specific Languages

Similar to EventReactor, domain-specific languages are defined as data types and are added to the GummyModule language. Each domain-specific language is defined via a specification and a so-called action class. The latter is a Java class implementing the compiler/runtime interpreter of the language. The former specifies the name of the domain-specific language, and the name of its action class. Listing 5 shows the structure of the linguistic construct `language` that is used to define the specification of domain-specific languages.

```
1 'language' <name> '{'
2     'action' '=' <class−name> ';'
3 '}'
```

**Listing 5.** The specification of domain-specific languages

Listing 6 shows an excerpt of the class `Action`, which is the super action class provided by gummy module. The method `compile`, which is executed when a program developed in GummyModule is compiled, must implement the functionality to compile the domain-specific program that is provided to this method via the parameter `code`.

The method `execute`, which is executed at runtime when the interpreter a reactor accepts an event, must implement the functionality to execute the domain-specific program at runtime when an event is provided for processing. This method receives the event to be processed and the runtime context as its argument; the runtime context contains references to the gummy module in which the program is defined and the enclosing gummy modules.

```
1 public abstract class Action {
2     public abstract void compile(String code) throws Exception;
3     public abstract void interpret(Event event, Context runtimecontext);
4 }
```

**Listing 6.** The super action class in GummyModule

### 1.5 Specification of Gummy Modules

In the GummyModule language, so-called gummy modules are the units of modularization. Listing 7 depicts the specification of gummy modules. As line 1 shows, a gummy module is specified by a `name`, which must be unique in the program. A gummy module is formed around four parts named as `required interfaces`, `provided interfaces`, `functions` and `bindings`. The GummyModule language assumes that the parts `functions` and `bindings` are encapsulated within a gummy module.

```
1  gummymodule ::= 'gummyodule' <name> '{'
2        requiredinterfaces?
3        providedinterfaces?
4        functions?
5        bindings?
6    '}'
7  requiredinterfaces ::= 'required interfaces' '{' ( <name> ':' <ri−type> (':='value)? ';' )∗'}'
8  providedinterfaces ::= 'provided interfaces' '{' ( <name> ':' <type> ';')∗'}'
9  functions ::= 'functions' '{'
10           ( <name> ':' <type> ( ':=' <value>)? ';' )∗
11           ( 'gummymodule' <name> ('language' ':=' <name> '{' <code> '}')? ')∗'}'
12 bindings::= 'bindings' '{' ( 'bind' '(' <fully−qualified−name> ',' <name>')' ';' )∗ '}'
13 ri−type ::= type | gummymodule−name
14 type ::= Java types '| event−type | event
15 value ::= −−a valid value
16 code ::= −− a valid piece of code
```

**Listing 7.** The specification of a gummy module

As line 7 shows, the required interfaces are defined using a `name` and a `type`. The types defined in the Java language, the defined event types, events and gummy modules are the supported types. As line 8 shows, the provided interfaces are also defined using a `name` and `type`; currently, the types defined in the Java language, the defined event types, and events are supported for the provided interfaces.

As lines 9 to 11 show, the functionality of a gummy module is defined in terms of a set of variable definitions and/or a set of in-lined primitive gummy modules. Each in-lined primitive gummy module is specified by a `name`, the name of the `language`, and a piece of `code` implemented in that language. In Listing 7, the parts `language` and `code` are defined as optional; as we will show it later, this gives the flexibility to configure a gummy module with different implementations.

As line 12 shows, using a set of binding expressions, the required/provided interfaces are bound to the implementations and/or variables defined within a gummy module. Since required interface can be a gummy module itself, it is possible to refer to its sub-gummy modules via their fully-qualified names.

There are two special kinds of gummy modules, i.e. `lifecycle` and `primitive`. A lifecycle gummy module implements the construction and destruction sub-concerns of an ordinary gummy module. A lifecycle gummy module is defined similarly to an ordinary gummy module, except that its provided interface must at least contain one event of the type `Activation` and one of the type `Deactivation`.

If the implementation of a gummy module is not in-lined inside the module, it can be defined as a `primitive` gummy module. Listing 8 shows the syntax of a primitive gummy module, which specifies a piece of program code and the language in which it is implemented. Primitive gummy modules do not have interfaces, consequently, they are not standalone. These modules must be used as part of the implementation of other gummy modules.

```
1 gummymodule ::= 'gummymodule' <name> 'is primitive' 'language' ':=' <name> '{'<code> '}'
```

**Listing 8.** The syntax of primitive gummy modules

### 1.6 Specification of Configurations

As listing 7 show, the GummyModule language enables identifying the sub-concerns of a gummy modules by means of unique identifiers. This facilitates configuring a gummy module by accessing its sub-concerns and modifying them. The GummyModule language offers a configuration language for this matter, whose syntax is depicted in Listing 9.

Here, a configuration script can contain a set of declarations and initializations. Within the block `declarations`, a set of variables are defined of the supported Java types or defined gummy modules. Within the block `initializations`, the previously these variables are initialized with desired values.

If the type of a variable is a gummy module, by means of fully-qualified names, it is possible to access the sub-concerns of that specific instance of the gummy module, and configure them. Currently, it is only possible to configure the implementation and the required interface a gummy module. If the required interfaces of two gummy modules jointly process one event, the order in which the event must be processed by these gummy modules is specified using the operator `precede` within the block `constraints`.

```
1  config ::= 'configurations' '{'
2     declarations?
3     initializations?
4     constraints?
5     '}'
6  declarations ::= 'declarations' '{'
7        ( <name> (',' <name> )* ':'
8              Java types | gummymodule−name| Event−type | Event ';' )*
9      '}'
10 initializations ::= 'initializations' '{'
11           ( <name>( '.' <name> )* ':=' <name> | value ';' )*
12    '}'
13 constraints ::= 'constraints' '{'
14       ('precede' '(' <gummymodule−name> ',' <gummymodule−name> ')' ';')*
15    '}'
```

**Listing 9.** The syntax of the configuration language

## 2 Runtime Event Processing in the GummyModule Language

The compiler of the GummyModule language maintains the information about the programmer-defined event types, gummy modules, their configuration and the order in which they must process events in a repository. The runtime environment of GummyModule keeps a reduced copy of this repository to access this information for processing the events.

Regardless whether the base program is concurrent, the current version of GuMoudle processes the events in a sequential synchronous manner similar to the way synchronous method invocations are executed in sequential programs. For this matter, the runtime environment keeps the incoming events in a queue, sorted by the timestap of the events. Listing 10 shows the algorithm for receiving and filtering an event.

The procedure `ProcessEvent` receives the event `e`, which is an event published by a base object or a gummy module. As lines 2 and 3 show, if `e` is of the type `Activation` or `Deactivation`, and the instance of the publisher gummy module is not already constructed or destructed, the runtime environment performs the construction or destruction, respectively. For other kinds of events,

runtime environment retrieves the information about the defined gummy modules from `repository` and makes use of the procedure `FilterEvent` to identify the required interfaces to which the event of interest matches. After the event is processed by the gummy modules of interest, if any, the flow of execution returns to the publisher of the event. The gummy modules may change the attributes of events, and when the flow of execution returns to the publisher, these changes must be applied to the publisher. The details about this is provided in [1].

The procedure `FilterEvent` receives the event `e` and the module `m` as its parameters, and identifies whether `e` matches any required interface `ri` of `m`. As line 11 show, if `ri` is a lifecycle sub-gummy module, the event `e` is recursively checked against the required interfaces of `ri`. As lines 12 and 13 show, if `ri` is an ordinary sub-gummy module, `e` is checked against the required interface of `ri` only if the instance of `m` is already constructed, or `ri` is a sub-gummy module of a lifecycle interface. This guarantees that none of the ordinary interfaces of `m` processes `e` before `m` is actually instantiated. The recursion finishes in lines 14 to 16, where `ri` is a primitive interface and the type of `e` matches the type of `ri`. In this case, if `m` is already instantiated, or `ri` belongs to a lifecycle interface, the event `e` is accepted, and the procedure `Bind` is invoked to bind the event to the functionality of `m`. It is note mentioning that GummyModule makes use of the stack `mstack` to keep track of the nested gummy modules against which `e` is checked.

```
1  procedure ProcessEvent (e: Event)
2    if (e.type == Activation and isConstructed(e.publisher) = false) then construct(e.pubisher);
3    else if (e.type == Deacivation and IsConstructed(e.publisher)) then destruct(e.pubisher);
4    else
5         for each (m in repository.modules) do FilterEvent(e, m);
6    //return flow to publisher
7  end procedure
8   FilterEvent (e: Event, m: GummyModule)
9    for each (ri in m.RequiredInterfaces) do
10     mstack.push(m);
11     if (IsLifecycle(ri)) then FilterEvent (e, ri);
12     else if (IsOrdinary(ri) and ((IsConstructed(m) or
13           mstack.contains(ModuleOfType(Lifecycle)))) then FilterEvent (e, ri);
14           else if (IsPrimitiveInterface(ri) and Matches(ri.type, e.type) then
15            if (IsConstructed(m) or mstack.contains(ModuleOfType(Lifecycle))) then
16              Bind (e, m);
17      mstack.pop();
18   end for
19 end procedure
```

**Listing 10.** Runtime Filtering of Events

Listing 11 shows the algorithm adopted by the procedure `Bind`, which receives the event `e` filtered by a required interface of `m`. For each binding `b` defined in `m`, lines 3 to 14 indicate that if `e` matches the source of the binding, and the target of binding matches a provided interface of `m`, the event `e` is bound to the provided

interface and is published outside `m` by invoking the procedure `Publish`. Lines 5 to 14 indicate if the target of the binding is a primitive gummy module, the piece of program specified as the primitive gummy module must be executed. Lines 8 to 12 indicate that if during the execution, the primitive module publishes an event, which matches a provided interface of `m`, the event must be published outside `m` by invoking the procedure `Publish`. After this event is processed, the flow of execution returns to line 11, which terminates the execution of the primitive module if the corresponding instance of `m` has been destructed as a result of processing this event.

The procedure `Publish` checks whether `m` that publishes `e` is a sub-gummy module. If so, it provides `e` to the module enclosing `m` so that it can be bound within the enclosing module. Otherwise, it publishes `e` to the runtime environment by invoking `ProcessEvent`.

```
1  procedure Bind(e: Event, m: GummyModule)
2    for each (b in m.Bindings) do
3      if (Matches (e, b.source)) then
4        if (Matches(b.target, m.ProvidedInterfaces)) then Publish(e, m);
5        else if (isPrimitiveModule(b.target)) then
6              do
7                  execute (b.target.program);
8                  out = b.target.published();
9                  if (Matches(out, m.ProvidedInterfaces)) then
10                 Publish(out, m);
11                 if (IsDestructed(m)) break;
12               end if
13               else discard(out);
14             until (b.target not terminated) do
15   end for
16 end procedure
17  Publish (e: Event, m: GummyModule)
18   if (isSubModule(m) = false) then ProcessEvent(e);
19   else Bind (e, m.enclosing);
20 end procedure
```

**Listing 11.** Runtime Binding of Events

## References


1. Malakuti, S.: Event Composition Model: Achieving Naturalness in Runtime Enforcement. PhD thesis, University of Twente (2011)
2. Malakuti, S., Aksit, M.: Evolution of Composition Filters to Event Composition. In: Proceedings of the 27th Annual ACM Symposium on Applied Computing. SAC '12, ACM (2012) 1850–1857
3. AspectJ: www.eclipse.org/aspectj/.
4. Compose*: http://composestar.sourceforge.net/.
5. AspectC: http://www.cs.ubc.ca/labs/spl/projects/aspectc.html.