Architecture Specification for Asynchronous Monitoring

Teng Zhang, Peter Gebhard, Oleg Sokolsky

July 2016

1 Overview

SMEDL is a specification language supporting both synchronous and asynchronous monitoring. Compared to synchronous monitoring, asynchronous monitoring can reduce the overhead by moving some heavy and non-critical checking to the monitors that run asynchronously with the target system. The whole monitoring system can be described as below:

$$MonitoringSystem = TargetSystem + MonitorNetwork$$

Monitor Network is defined as a set of monitor instances connecting with each other by events using asynchronous communication. Moreover, a lot of properties will involve the values, which brings the necessity of parametric monitoring: multiple instances of a monitor may be created dynamically during the execution. Fig 1 illustrates an example of monitoring system containing one instance of synchronous monitor $SyncMon$ and two instances of asynchronous monitor $AsyncMon$.

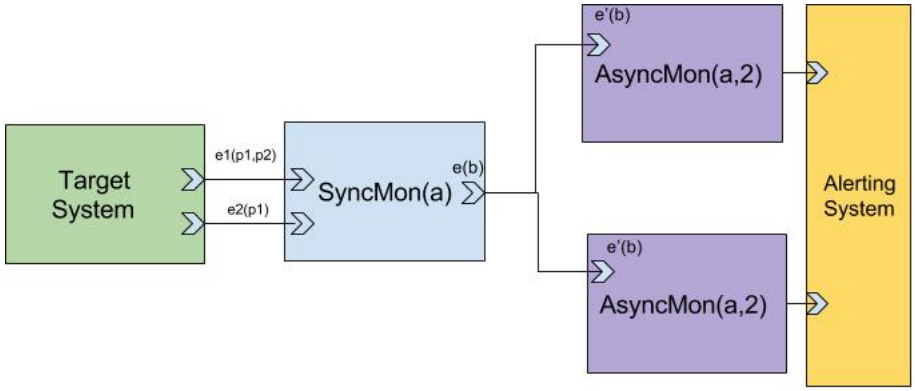

Figure 1: An example of Monitoring System

Then problem becomes how to route events between dynamically created monitor instances. However, even with dynamic instantiation, any run-time configuration can be described statically using patterns. To specify patterns of event connections between monitor instances, an architecture description language for SMEDL(A4SMEDL) is proposed. Section 2 will present the concrete syntax of the language while the abstract syntax will be proposed in Section 3. Then, the semantics based on the static instantiation of monitors will be given in Section 4. Section 5 will propose how to implement asynchronous communication using a messaging middleware rabbitMQ.

2 Concrete Syntax

A4SMEDL is used to design the architecture of the monitor network by specifying the event connections among monitors. For now, each monitor will be compiled separately into C files and the A4SMEDL file will be taken as input of the compiler for each compilation to generate code for event connection and communication. Since each monitor specification will not know other monitors, it is necessary to declare interface of each monitor besides the architecture specification, just as the concrete illustrated below. In the monitor declaration, user can specify the type of monitor, synchronous(Sync) or asynchronous(Async). Note that if a monitor can receive the events from other monitor, it must be declared as asynchronous monitor. If no type is specified, the monitor will be defined as asynchronous monitor by default. The interface of a monitor includes the parameter list of the monitor, imported event list and exported event list. Note that only the type of parameters are needed.

```
Top ::= 'System' identifier ':='
    {monitorDeclaration}+
    ArchitectureSpec
monitorDeclaration ::= ['Async' | 'Sync' ] identity
'('parameter_list')'
'{'
    'imported' event_definition_list
    ['exported' event_definition_list]
'}'
event_definition ::= ['error'] identifier ['('
parameter_list ')']
parameter_list ::= type {',' type}* | ()
```

The architecture specification part contains a set of connection expressions specifying the event connections between monitors. A connection expression has four parts: an option channel name, source, target and pattern specification. If two connection expressions have the same channel name, the target should be same. If no channel name is specified, the default channel name will be taking the form "$sourceMonitorName - sourceEventName$". $Source$ and $target$

both take the form $MachineName.eventName$. For example, "M1.e1⇒M2.e2" means exported event e1 of monitor M1 is connected to the imported event e2 of monitor M2. In SMEDL, several monitor instances can be instantiated from one monitor object. In order to specify how instances are connected, pattern specification is introduced. The pattern specification is a set of pattern expressions. A pattern expression takes the form *term operator term* where the *operator* is equal('=') or not equal('<>'). *Term* can be parameters of the monitor, attributes of the event or the literal value. The parameter of the monitor is defined as 'M[i]', standing for the i^{th} parameter of the monitor M. The attribute of the monitor is defined as 'e[i]', standing for the i^{th} attribute of the event e. Note that whenever $M.e$ is used in a connection expression, monitor declarations should include M, and the declaration for M should include e as an imported event.

ArchitectureSpec ::= {ConnectionExpr} *
ConnectionExpr ::= [Identifier ':']
MachineName.eventName '=>' MachineName.eventName
'{'PatternSpec'}'
MachineName | *eventName* ::= identifier
PatternSpec ::= () | PatternExp {';' PatternExp}*
PatternExp ::= Term Operator Term
Operator ::= '=' | '<>'
Term ::= MachineName'['Index']' | eventName
'['Index']'
Index ::= natural number

Here we give some examples to illustrate how A4SMEDL describe the architecture patterns and how these patterns relate to the dynamic architecture of monitor networks. Note that the target system is omitted from these examples.

Example 1: One to One Pattern

There are three kinds of monitors A, B and C where A sends event e to B as e' and B sends e" to C as e". In this example, we want to ensure that instances of A only send events to instances of B with the same identifier, so as the case between B and C. The A4SMEDL specification is shown below. The first three blocks specify the interface of A, B and C, which follows two pattern specifications *ch1* and *ch2*. For ch1, event e of A is connected to e' of B with the pattern expression $A[0] = B[0]$ indicating that the first parameter of A equals to the first parameter of B. Ch2 has the similar structure, connecting e" of B to e"' of C with the same parameter.

```
System ABC :=
    A(int){
      exported e();
    }
    B(int){
      imported e'();
      exported e"();
    }
    C(int){
      import e"'();
    }
    ch1:A.e => B.e'{A[0]=B[0]}
    ch2:B.e" => C.e"'{B[0]=C[0]}
```

Monitor instances can be created statically by instrumentation or dynamically during the execution. However, the dynamical architecture of the monitor network will always be consistent with the specification. Fig 2 shows an possible dynamic structure complying with the specification given above. It illustrates that only instances with the same identifier will connect with each other through event connection.

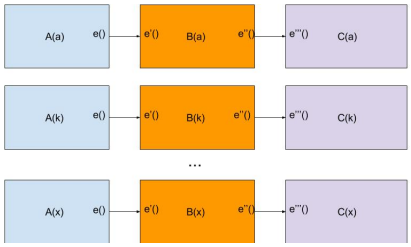

Figure 2: An example of one to one pattern

Example 2: Multiple to One Pattern

The following example shows the multiple to one pattern between monitor instances. In a multi-threaded web system, monitor A will count the number of API called. Monitor will count overall API calls from one server by collecting count from instances of A. The A4SMEDL specification is shown below. A has two parameters of which the first one represents the ID of the server and the second one is the API ID. B has one parameter representing the ID of the server. The pattern specification connects e of A to e' of B with the same first parameter value.

```
System AB :=
    A(int,int){
      exported e();
    }
    B(int){
      imported e'();
    }
    ch1:A.e => B.e'{A[0]=B[0]}
```

In the actual system, there will multiple instances of A and B, as shown in Fig 3. However, as the figure indicates, instance of B will only collect data from instances of A with first parameter matched.

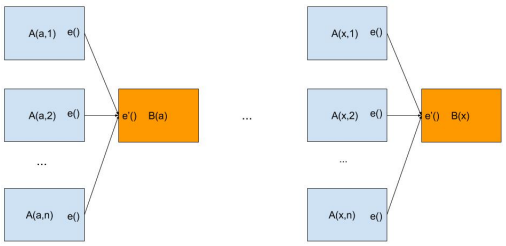

Figure 3: An example of one to multi pattern

Example 3: Dynamic Routing Pattern

Example 3 is the reverse of example 2. Assume a monitor A receives all kinds of API calls from a server, we want to count calls according to the type of API. The A4SMEDL specification is shown below. The parameter of A is the server ID. B has two parameters: the first one is the server ID and the second one is type of the API that the monitor will count. Event e has one attribute representing the type of API which needs to match the second parameter of B, as shown in the pattern expression.

```
System AB' :=
    A(int){
      exported e(int);
    }
    B(int,int){
      imported e'(int);
    }
    ch1:A.e => B.e'{A[0]=B[0];e[0]=B[1]}
```

Fig 4 illustrates dynamic event connection between monitor A and B. On the left part, the attribute of e is 1 so that A(m) sends the event e(1) to B(m,1) while on the right part, A(m) sends e(2) to B(m,2).

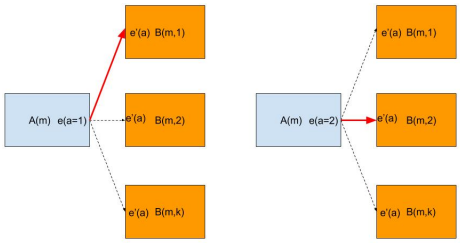

Figure 4: An example related to event attribute

3 Abstract Syntax

Here we give the abstract syntax of A4SMEDL which will be used to define semantics of monitor network in the next section. The structure follows concrete syntax which is consist of two parts: declaration of monitor types involved and the architecture specification, as shown below. The *MonitorType* restricts the scope of monitor and event types in the architecture specification. *ImportedEvents* and *ExportedEvents* will always be used respectively as the target events and source events in the event connection.

System ::= <MonitorType, ArchitectureSpec>
MonitorType ::= <Type, Name, ParameterList,
ImportedEvents,ExportedEvents>
Type ::= Async | Sync
ImportedEvents | ExportedEvents ::= Set of EventTypes
EventType ::= <Name, ParameterList>

The architecture specification is shown below.

ArchitectureSpec ::= Set of ConnectionExprs
ConnectionExpr ::= <Id, SourceMonitor, SourceEvent, TargetMonitor,
TargetEvent, PatternSpec>
PatternSpec ::= Set of PatternExprs
PatternExpr ::= <Operator, LeftTerm,RightTerm>
Operator ::= Equal | NotEqual
LeftTerm|RightTerm ::= Term
Term ::= <TermType, Index>
TermType ::= MonitorType.Name | EventType.Name

Although provided with flexible syntax, one needs to write well-formed architecture specification to build a monitor network. More specifically, following restrictions need to be followed.

- All monitors and events should be within the scope provided in *MonitorType*.

- Interface of source event and target event should match with each other.

The restrictions on *PatternSpec* and *PatternExpr* are listed here.

- Scope of PatternExpr is SourceMonitor, SourceEvent, TargetMonitor in the ConnectionExpr it belongs to.
- Term from SourceMonitor and TargetMonitor shall appear exactly once in *PatternExpr*.

To describe the semantics in a more convenient way, we assume that left hand side of *PatternExpr* will be for the target monitor while right hand side will be for the source monitor.

4 Semantics under static instantiation

An A4SMEDL specification gives a pattern for event connections between monitor instances in the monitor network. This section presents the semantics of monitor network with fixed monitor instances, which means given a source monitor instance and one of its event, how to get the target instance(s) and event to which the source event will be delivered.

A fixed monitor network has a set of monitor instances, a set of imported events, a set of exported events, an architecture specification and a routing function(*Routing*) describing the event connections between monitor instances. *Routing* is a partial function mapping an exported event of a monitor instance to a set of imported events of monitor instances based on a $ConnExpr$.

MonitorNetwork ::= <MonitorInstances, ExportedEventInstances, ImportedEventInstances, ArchitectureSpec,Routing>
ExportedEventInstance | ImportedEventInstance ::= EventInstance
MonitorInstance ::= <MonitorType, ParameterBinding>
EventInstance ::= <EventType,AttributeBinding>
AttributeBinding|ParameterBinding ::= Binding
Binding ::= $N \rightarrow V$
Routing ::= MonitorInstances $\rightarrow$ ExportedEventInstances $\rightarrow$ ArchitectureSpec $\rightarrow$ P(MonitorInstances*ImportedEventInstances)

Here we give the definition of *Routing* assuming that it takes source monitor instance $SMon$, a source event from the instance Se and a ConnectionExpr Ce defined as $(Name, SMon, Se, TMon, Te, PatternExprs)$ as the input and returns the set of pairs $Target$.

$$Target = Routing(SMon, Se, Ce)$$

Before defining *Routing*, we need to take a detailed look at $PatternExprs$. In $PatternExprs$, it is reasonable to define multiple equations with the same

target term such as $\{B[0] = A[0]; B[0] = e[0]\}$ where A and B are source and target monitor and e is the exported event of A connecting to imported event of B. The $PatternExprs$ implicitly indicates that $A[0] = e[0]$, which is actually a predicate on the source monitor and event. Here we assume that $PredSource$ is the set of all predicates on the source side, which can be easily constructed from $PatternExprs$.

Routing is defined below. First, we need to check if all predicates in $PredSource$ are satisfied. If not, there will be no target monitor instance. Then For each pair $(u, e') \in Target$, it should satisfy a conjunction of predicates: u will be of type $TMon$; e' will be of type Te; for each parameter u[i] of u, if it does not appear in $Ce.PatternExprs$, there will be no corresponding predicate added; if there exists a $PatternExpr$ specifying TMon[i] equals to a value x, a predicate $u[i] = x$ will be added; if there exists a $PatternExpr$ specifying TMon[i] is not equal to a value x, a predicate $u[i] \neq x$ will be added. According to the syntax, value x could be a parameter of the source monitor instance or an attribute of the source event.

$$Routing(SMon, Se, Ce) = \begin{cases} empty, & \neg(\wedge_{pred \in PredSource} pred) \\ \{(u, e') | u.MonitorType = TMon \wedge e'.EventType = Te & \\ \wedge_{0 \leq i \leq PNum(TMon)} UP(i)\}, & (\wedge_{pred \in PredSource} pred) \end{cases}$$

where $PNum(TMon)$ returns the number of parameters of $TMon$ and

$$UP(i) = \begin{cases} empty & , \forall expr \in PatternExprs, expr.LeftTerm \neq TMon[i] \\ u[i] = x & , \exists expr \in PatternExprs, expr = (Equal, (TMon, i), x) \\ u[i] \neq x & , \exists expr \in PatternExprs, expr = (NotEqual, (TMon, i), x) \end{cases}$$

5 Implementation

RabbitMQ is used as the messaging middleware to implement the asynchronous communication between monitors. We first introduce some basic concepts in RabbitMQ, then present how to implement message sending using RabbitMQ.

RabbitMQ is an open-sourced messaging broker for asynchronous communication among applications, which supports multiple communication patterns such as work queues, pub/sub, routing and topics. In the current context, topic pattern will be used. In topic pattern, there are four roles, producer, consumer, queue and exchange. Producer and consumer will respectively produce and consumer messages. In SMEDL, synchronous monitors will only produce messages while asynchronous monitors may both produce and consumer messages. Queue is used to save and forward messages. Each consumer will have its own one unique queue. In RabbitMQ, producer never sends messages directly to the queue. Instead, messages will be first sent to exchange which will further send them into corresponding queues.

In RabbitMQ, a message will be wrapped as an envelope sent to the exchange. Envelope will not only contain the content of the message but also the routing key indicating to where the message should be delivered. The routing key takes the form of a list of words delimited by dots like "M1.e2". Max length of routing key for now is 255 bytes. On the queue side, each queue can have several binding keys. Exchange will send the message to the corresponding queues with at least one binding key that can match the routing key of that message. The binding key takes the similar form, but usually it represents a pattern so that some wildcards are used: *(star) can substitute one word while #(hash) can substitute any number of words.

Here the format of routing key and binding key are defined informally as below. In A4SMEDL, each *PatternSpec* corresponds to a channel. For each channel, a binding key will be created and bound to the queue from which the monitor instance will get events. Without loss of generality, we use a *PatternSpec* to illustrate how the binding key is generated. Assume M1 has a parameters, M2 has b parameters and e has c attributes. PatternSpec *spec* is defined as

$$ch1 : M1.e1 => M2.e2\{M1[k_1] = M2[m_1], ..., M1[k_i] = M2[m_i], e1[w_1] = M2[v_1], ..., e1[w_j] = M2[v_j]\} \text{ where } k_1 < k_2 < ... < k_i < a \text{ and } v_1 < v_2 < ... < v_j < c.$$

The corresponding binding key will be

$$ch1.x[1].x[2]....x[a].e1.y[1].y[2]....y[c] \text{ where}$$

$\forall i \in [0, a]$,

$$x[i] = \begin{cases} * \ , \forall l \ k_l \neq i \\ M2[m_l] \ \exists l, k_l = i \end{cases}$$

and
$\forall i \in [0, c]$,

$$y[i] = \begin{cases} * \ \forall l \ , w_l \neq i \\ M2[v_l] \ \exists l, w_l = i \end{cases}$$

Routing key takes a similar form as the binding key except that every x[i] will be M1[i] and y[i] will be e1[i]. The message body will contain the event information, taking the form

$$e/x[1]/x[2]/...x[m]$$

where e is the name of source event and x[k] is the k^{th} attribute of event e.

In the example above, we assume all k have different value, so do all m and v. If it is not the case, further predicate checks need to be done at the target monitor instance. For instance, if we have $M1[k_1] = M2[m_1]$ and $M1[k_2] = M2[m_1]$, then we need to ensure $M1[k_1] = M1[k_2]$. Moreover If the operator of the pattern expression is not equal, the binding key will not have corresponding expression. The further filtering will also be done.

Although concrete implementation is not shown here, by scanning the architecture specification written in A4SMEDL and the SMEDL file, the compiler will be able to generate communication code. For the synchronous monitor, only one connection related to the message sending is needed while for the asynchronous monitor, two connections are needed, one for sending message and one for receiving messages. For now, all monitor instances will be used one common exchange and each monitor instance receiving the message will have its own queue where each related connection specification corresponds to one binding key. When the monitor tries to send the message, the message will be put into an envelope object with both message and the routing key. The exchange will send the envelope to the proper queue where the consumer monitor instance will grab the message once a time . Further filtering may be done if there are pattern expressions involving the unequal operator.