Valour User Manual

Introduction

What is Valour ?

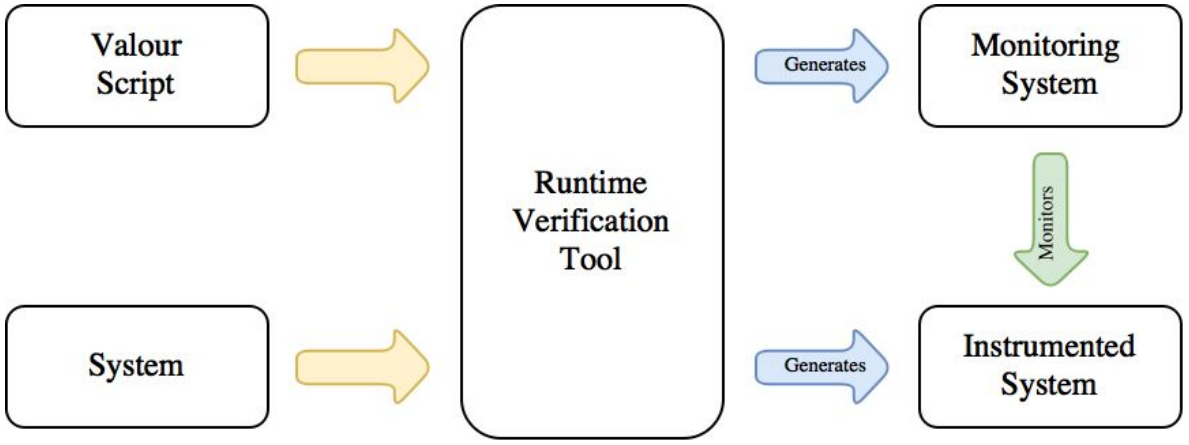
Valour is a language that allows users to define rules that can be used to monitor a system at runtime. These rules will allow the user to define what circumstances of interest happening in the system will trigger events to be monitored, and how to react to these events. The rules may react to these events by maintaining some form of state about the system, notifying the system (or some alerting module) about specific situations, and performing some actions in order to influence the system's operation.

Runtime Verification Tool

Defining the rules of how a system should be monitored is by itself not useful unless the system can be monitored and the rules executed. This gap is filled by the Runtime Verification Tool which given a Java-based system to be monitored and a Valour script, it will generate two new artifacts:

1. **Monitoring System** - An executable instantiation of the monitoring rules defined in the Valour script. The Monitoring System will listen to system events, evaluate monitoring rules, maintain the required state, and execute any actions specified in the Valour script.
2. **Instrumented System** - A version of the System (to be monitored) that has been instrumented to interact with the Monitoring System. Instrumentation of the System will happen at compile time through AspectJ (with the possibility of using an agent to weave at runtime), and will generate the necessary events as defined in the Valour script.

The diagram below depicts how supplying a Valour script and a System as input to the Runtime Verification Tool, a Monitoring System and an Instrumented System are created whereby the Monitoring System is configured to monitor the Instrument System.



Example

In order to better introduce Valour, this section presents a simple system, a set of rules that monitor the system's behaviour, and the resulting Valour script to fulfil the monitoring requirement. The set of rules that monitor the system will be modified incrementally in order to showcase different features of the Valour language.

Note that the scope of this example is to merely introduce the reader to Valour and does not aim to present an exhaustive example and description of all of Valour's features. The individual features will be presented in the next sections starting with a basic description of the **Valour Language Structure** followed by detailed explanations of the three major parts of the language - **Imports**, **Declarations**, and **Rules**. Finally, the appendix presents the full BNF of the Valour language.

Simple System

For this example, we will assume a system whereby users can login, logout and perform some actions which are beyond the scope of this example. A user may be a normal user, or may have a privileged status of being a gold user.

Rules

The system operates using the following simple rules

- Only 110 users may be logged in the system at any given time. If this condition is not met, then the monitoring system should generate an alarm.
- Only 10 normal users may be logged in the system at any given time. If this condition is not met, then the monitoring system should generate an alarm.
- Only 100 gold users may be logged in the system at any given time. If this condition is not met, then the monitoring system should generate an alarm.
- A normal user may only submit €100,000 in payments during a given day. If this condition is not met, then the monitoring system should generate an alarm.
- If the system is shutdown, then the monitoring system should generate an alarm.

Valour Script

Instead of presenting the final Valour script at one go, the resulting Valour script will be presented in incremental enhancements in order to introduce the different features provided by Valour.

Capturing the events

The first step is to declare the events of interest. In this example, the events will be triggered by an AOP call pointcut expression to be applied on the system. More information about AOP can be found in the appendix, but at this point it is enough to know that an event will be generated exactly before the defined method is invoked. It is assumed that the system provides the following methods:

- LoginHandler.successfulLogin(User u) - invoked whenever a user successfully logs in.
- LoginHandler.successfulLogout(User u) - invoked whenever a user successfully logs out.

- TransactionHandler.submitTransfer(User u, String toAccount, BigDecimal amount) - invoked in order to submit a transfer.
- SystemHandler.shutdown() - invoked whenever the system is shutdown.

```
import com.example.model.User;

declarations {
	event userLogin() = {
		system controlflow trigger LoginHandler.successfulLogin(User u)
	}

	event userLogout() = {
		system controlflow trigger LoginHandler.successfulLogout(User u)
	}

	event transferSubmitted() = {
		system controlflow trigger TransactionHandler.submitTransfer(User u,
		String toAccount, BigDecimal amount)
	}

	event shutdown() = {
		system controlflow trigger SystemHandler.shutdown()
	}
}
```

Note how the example defines four events, associating the events with the respective method invocation happening in the system. Also, note that all of the event decelerations are happening within a *declarations* block.

The example also introduces the use of *import*s, where the User class is being imported from system's code base. In this way, any mention of the class User in the Valour script does not need to be fully qualified.

Creating the first rule

Given that the events have been defined, we next start defining the monitoring rules. We will start by defining the rule which specifies that whenever the system is being shutdown, then an alarm should be generated.

A Rule can be defined as a triple of <Event, Condition, Action> i.e. **On** an <Event> **If** the <Condition> are satisfied **Then** perform <Action, and the syntax for defining a rule is: <Event> | <Condition> -> <Action>. Note that conditions are optional and may be omitted.

The below snippet shows the example above modified to have the rule necessary rule created.

```
import com.example.model.User;
import com.example.alarms.AlarmHelper;

declarations {
	event userLogin() = {
		system controlflow trigger LoginHandler.successfulLogin(User u)
	}

	event userLogout() = {
		system controlflow trigger LoginHandler.successfulLogout(User u)
```

```
        }

        event transferSubmitted() = {
                system controlflow trigger TransactionHandler.submitTransfer(User u,
                String toAccount, BigDecimal amount)
        }

        event shutdown() = {
                system controlflow trigger SystemHandler.shutdown()
        }

}

shutdown() -> { AlarmHelper.generateAlarm("System being shutdown"); }
```

Note how the rule states that on the shutdown event then the AlarmHelper should be instructed to generate an alarm. The AlarmHelper is a helper method defined in the system to generate an alarm.

Creating the first action

In common to all the rules is the action to generate an alarm whenever a rule is broken. Valour allows the creation of an action that can be used in multiple rules. In the below code snippet, the previous example has been modified in order to define the new action `generateAlarm(String message)`. Note that the unmodified part of the code is being presented in grey.

```
import com.example.model.User;
import com.example.alarms.AlarmHelper;

declarations {
        event userLogin() = {
                system controlflow trigger LoginHandler.successfulLogin(User u)
        }

        event userLogout() = {
                system controlflow trigger LoginHandler.successfulLogout(User u)
        }

        event transferSubmitted() = {
                system controlflow trigger TransactionHandler.submitTransfer(User u,
                String toAccount, BigDecimal amount)
        }

        event shutdown() = {
                system controlflow trigger SystemHandler.shutdown()
        }

        action generateAlarm(String message) = {
                AlarmHelper.generateAlarm(message);
        }
}

shutdown() -> { #generateAlarm("System being shutdown"); }
```

Adding state for rules

So far, the only rule that has been defined does not require any state to be maintained. However, the other rules all require some kind of state to be maintained. We start by adding the rules for the total number of users logged in the system.

```
import com.example.model.User;
import com.example.alarms.AlarmHelper;

declarations {
	event userLogin() = {
		system controlflow trigger LoginHandler.successfulLogin(User u)
	}

	event userLogout() = {
		system controlflow trigger LoginHandler.successfulLogout(User u)
	}

	event transferSubmitted() = {
		system controlflow trigger TransactionHandler.submitTransfer(User u,
		String toAccount, BigDecimal amount)
	}

	event shutdown() = {
		system controlflow trigger SystemHandler.shutdown()
	}

	action generateAlarm(String message) = {
		AlarmHelper.generateAlarm(message);
	}
}

shutdown() -> { #generateAlarm("System being shutdown"); }

state {
	Integer users = {{ 0 }}
}
in {
	//rule to increment counter on login
	userLogin() -> { users++; }
	//rule to decrement counter on logout
	userLogout() -> { users--; }
	//rule to trigger alarm
	userLogin() | { return (users > 110) } -> { #generateAlarm("Too many users");}
}
```

The number of users currently logged in is being maintained in the *state* block. As part of the state block three rules are defined that,

1. Increment the counter on a user login event.
2. Decrement the counter on a user logout event.
3. Trigger an alarm if a successful user login is detected when the total number of logged in users is greater than 110. Note how this last rule is protected by a condition.

This extension to the example introduces two new features in Valour - the {{}}-notation and the {}-notation for evaluating values.

Using the {{}}-notation, the user may define a single statement that evaluates to a value of the expected type. In the example, this notation has been used to initialise the value of `users`.

Using the {}-notation, the user may define a code block that is expected to end with a return statement that returns a value of the expected type. In the example, this notation has been used to return the value for the condition to guard the rule action execution. As expected, the return type for a condition is boolean, and so the example in returning the evaluation of `(users > 110)`.

Creating conditional events

Valour allows the creation of events only if a certain condition is satisfied. Our example will be extended to use this feature in order to define the login events for gold users and normal users. Note how the *when*-clause is used to apply a filter on the events. The example will also be extended in order to add the rules associated to the gold user and normal user logins/logouts.

```
import com.example.model.User;
import com.example.alarms.AlarmHelper;

declarations {
	event userLogin() = {
		system controlflow trigger LoginHandler.successfulLogin(User u)
	}

	event userLogout() = {
		system controlflow trigger LoginHandler.successfulLogout(User u)
	}

	event normalUserLogin() = {
		system controlflow trigger LoginHandler.successfulLogout(User u)
		when {{ !u.isGoldUser() }}
	}

	event normalUserLogout() = {
		system controlflow trigger LoginHandler.successfulLogout(User u)
		when {{ !u.isGoldUser() }}
	}

	event goldUserLogin() = {
		system controlflow trigger LoginHandler.successfulLogout(User u)
		when {{ u.isGoldUser() }}
	}

	event goldUserLogout() = {
		system controlflow trigger LoginHandler.successfulLogout(User u)
		when {{ u.isGoldUser() }}
	}

	event transferSubmitted() = {
		system controlflow trigger TransactionHandler.submitTransfer(User u,
		String toAccount, BigDecimal amount)
	}

	event shutdown() = {
		system controlflow trigger SystemHandler.shutdown()
	}

	action generateAlarm(String message) = {
		AlarmHelper.generateAlarm(message);
	}
}
```

```
shutdown() -> { #generateAlarm("System being shutdown"); }

state {
        Integer users = {{ 0 }}
        Integer normal = {{ 0 }}
        Integer gold = {{ 0 }}
}
in {
        //rules to increment counters on login
        userLogin() -> { users++; }
        normalUserLogin() -> { normal++; }
        goldUserLogin() -> { gold++; }

        //rules to decrement counters on logout
        userLogout() -> { users--; }
        normalUserLogout() -> { normal--; }
        goldUserLogout() -> { gold--; }
        //rules to trigger alarms
        userLogin() | { return (users > 110) } -> { #generateAlarm("Too many users");}
        normalUserLogin() | {{normal > 10}} -> { #generateAlarm("Too many normal
        users");}
        goldUserLogin(u) | {{gold > 100}} -> { #generateAlarm("Too many gold users");}
}
```

Replicating state for rules

Valour allows for state to be replicated for particular domain objects being created, allowing the implementation of rules that talk about a single domain object. For instance, the rules of our example system state that, "A normal user may only submit €100,000 in payments during a given day." This rule is talking about localised state for every user.

In order to be able to correctly create a monitor for this rule, we need to be able to maintain the number of transactions submitted by each user. We will introduce three Valour features that will allow us to maintain state per user.

1. **Categories** - The first thing that is required by Valour is the definition of the categories i.e. the domain objects of interest for which the user would like to maintain a separate state. For each category, the user is required to define an index type which will allow the RV tool to uniquely identify the category instance being managed. Category statements, start with the *category* keywords and are placed in the *declaration* block.
2. **Event Categorisation** - Once categories are defined, then the script writer may define which categories events belong to. For instance, in our example we are talking about transfers in relation to a particular user. So, given a user category and a transfer event we may say that a transfer event belongs to a user category instance if the transfer was submitted by that user. This relationship is defined by using the *belonging-to* clause within the event declaration.
3. **State Replication per Category** - Finally, we can define state variables to be replicated per particular categories and define rules which are bound to this state replication. This is done using the *replicate-foreach* construct to be presented in the example.

The extended example is shown below. Note how the `transferSubmitted` event has been modified to have two parameters - the user ID and the transaction amount. These parameter are obtained from the AOP call pointcut expression in the following way,

- usedId - The where clause in the event declaration allows the extraction of data from the AOP call pointcut expression. In this case, we are extracting the userId from the user.
- amount - Obtained directly from the AOP call pointcut expression. Note that Valour will automatically use any parameters that match by name.

```
import com.example.model.User;
import com.example.alarms.AlarmHelper;
import java.util.Date;
import org.apache.commons.lang.time.DateUtils;

declarations {
	category USER indexed by Long

	event userLogin() = {
		system controlflow trigger LoginHandler.successfulLogin(User u)
	}

	event userLogout() = {
		system controlflow trigger LoginHandler.successfulLogout(User u)
	}

	event normalUserLogin() = {
		system controlflow trigger LoginHandler.successfulLogout(User u)
		when {{ !u.isGoldUser() }}
	}

	event normalUserLogout() = {
		system controlflow trigger LoginHandler.successfulLogout(User u)
		when {{ !u.isGoldUser() }}
	}

	event goldUserLogin() = {
		system controlflow trigger LoginHandler.successfulLogout(User u)
		when {{ u.isGoldUser() }}
	}

	event goldUserLogout() = {
		system controlflow trigger LoginHandler.successfulLogout(User u)
		when {{ u.isGoldUser() }}
	}

	event transferSubmitted(Long userId, BigDecimal amount) = {
		system controlflow trigger TransactionHandler.submitTransfer(User u,
		String toAccount, BigDecimal amount)
		where userId = {{ u.getId() }}
		belonging to USER with index {{ u.getId() }}

	}

	event shutdown() = {
		system controlflow trigger SystemHandler.shutdown()
	}

	action generateAlarm(String message) = {
		AlarmHelper.generateAlarm(message);
	}
}

shutdown() -> { #generateAlarm("System being shutdown"); }

state {
	Integer users = {{ 0 }}
	Integer normal = {{ 0 }}
```

```
        Integer gold = {{ 0 }}
}
in {
        //rules to increment counters on login
        userLogin() -> { users++; }
        normalUserLogin() -> { normal++; }
        goldUserLogin() -> { gold++; }

        //rules to decrement counters on logout
        userLogout() -> { users--; }
        normalUserLogout() -> { normal--; }
        goldUserLogout() -> { gold--; }
        //rules to trigger alarms
        userLogin() | { return (users > 110) } -> { }
        normalUserLogin() | {{normal > 10}} -> {#generateAlarm("Too many normal users");}
        goldUserLogin(u) | {{gold > 100}} -> { #generateAlarm("Too many gold users");}

        replicate {
                Date lastTxDate;
                BigDecimal dayTx = {{ BigDecimal.ZERO }}
                BigDecimal limit = {{ new BigDecimal("100000") }}
        }
        foreach USER u {
                transferSubmitted(userId, amount) ->
                        {
                                Date today = new Date();

                                if (
                                        (#lastTxDate == null)
                                        ||
                                        (!DateUtils.truncatedEqualsTo(today, #lastTxDate,
                                        Calendar.DATE))
                                ) {
                                        #lastTxDate = today;
                                        #dayTx = BigDecimal.ZERO;
                                }

                                #dayTx = #dayTx.add(amount);

                                if (#dayTx.compareTo(#limit) > 0){
                                        String msg = "User " + #userId + " exceeded transfer
                                        limit";
                                        #generateAlarm(msg);
                                }
                        }
        }

}
```

Valour Language Structure

The Valour language may be defined by describing it in three distinct parts.

1. **Imports** - The imports section is equivalent to the imports section found in a Java class i.e. its purpose is to declare types to be used in the Valour script avoiding the need to fully qualify the types at every use.
2. **Declarations** - A declaration allows the user to declare definitions that can be reused throughout the script. These declarations can be of type **event**, **category**, **condition**,

and **action**. Each of these will be explained in further detail (with examples) in the following sections.

3. **Rules** - Rules are the heart of Valour script, and define the rules to be followed while monitoring a system. A Rule can be defined as a triple of <Event, Conditions, Actions> i.e. **On** an <Event> **If** the <Conditions> are satisfied **Then** perform <Actions>. Supporting the Rules are constructs that allow for creation of local state (e.g. total number of observed logins), and creating replicated state for each instance of a domain object.

The different parts are put together to obtain the Valour script structure as shown below.

```
<import> *

<declarations-block>?

<rule> *
```

Describing the above in text, a Valour script starts off with zero or more import statements. The imports are followed by an optional declarations block which may contain any number and type of declarations (to be explained later). Finally, the script may contain a number of rules that specify the monitoring rules. Note that in turn, a rule may be a basic or a complex rule, and complex rules have complex structures which will be described later.

Though the Valour language structure is being described using these three distinct parts, it is important to note the close relationship shared between declarations and rules. Not only do declarations define blocks that can be referenced in the rules, but also some rules allow new declarations to be defined within their own body.

The following sections will delve into each of these different parts, explaining its structure and giving examples of how a Valour script may be built.

Imports

Valour allows the script writer to reference classes from the system's code base. As will be discussed later in the document, the script writer might want to reference the system's code base to define a category index, evaluate some condition or perform a required action.

In order to avoid having to fully qualify the types, Valour allows the script writer to import the class in the beginning of the Valour script and later simply reference the class using its simple name. The import section will be very familiar to anyone with basic Java development experience as it is equivalent to the imports section found in Java classes.

The basic structure of an import statement is shown below.

```
import <fully-qualified-class-name>;
```

The following is an example of an import section that is import three classes. In this way, the Account, Transaction and User class can now be referenced within the Valour script using their simple name instead of the fully qualified name.

```
import com.example.valour.Account;
import com.example.valour.Transaction;
import com.example.valour.User;
```

Declarations

Declarations allow the user to define declarations that can be reused across the script - in other declarations and in rule specifications.

A declaration block is enclosed within curly brackets and marked with the *declarations* keyword. The basic syntax of a declaration block is shown below.

```
declarations {
        <declaration>*
}
```

Where the declaration can be a **category**, an **event**, an **condition**, or an **action**.

Note that a declaration block may be defined within the body of a complex rule and thus the declarations might have a restricted scope rather than be accessible globally. Nesting of a declarations block with a complex rule will be discussed in detail later on in the document.

The remainder of this section will describe the different type of declarations available, and how they can be employed to build valid and useful Valour monitoring scripts.

Categories

A category is an entity of interest in the Valour script, and typically categories correspond to domain objects defined in the system e.g. User, Account, Transaction etc. In order for the RV Tool to correctly identify different instances within a category, the user is required to specify an index type. In other words, were the RV Tool to put all instances of a category in a map, the specified index type would define the key type to be used in the map. The declared categories will be used in the event definitions to define how an event is related to the defined categories, and rules can then be defined to maintain state per category of the event.

The basic syntax for defining a category is shown below.

```
category <category-name> indexed by <java-type>
```

A number of examples are presented below, creating the following categories:

1. USER: In a given system, the script writer would like to be able to identify events happening for individual users in order to maintain user specific state such as number of failed logins per user. The unique identifier for users in the system is a database auto-increment ID of type Long, hence the USER category is indexed by Long.
2. SESSION: In a given system an HTTP session is identified by a UUID. Using this category, the script writer may now maintain statistics about every session including session start time, and total number of requests serviced through the session. This example also shows how a fully qualified class may be used as the category index type.
3. TRANSACTION: In a given system, a money transfer is represented as a transaction where every transaction is uniquely identified by an `org.example.transaction.TransactionId` class instance. Using this category, the script writer may now maintain state about every transaction such as submission time, status, and approval time. This example also shows how a fully qualified class obtained from the system's code base may be used as the category index type.
4. ACCOUNT: In a given system, transactions occur between accounts and accounts are identified by an `org.example.transaction.AccountId` class instance. Using this category, the script writer may now maintain state about every account such as the average number of transactions per day, and the average transaction amount.

Note that as a convention, the category name should be defined in upper case.

```
import com.example.account.AccountId;

category USER indexed by Long

category SESSION indexed by java.util.UUID

//referencing a fully qualified system class
category TRANSACTION indexed by org.example.transaction.TransactionId

//referencing a system class that has been imported
category ACCOUNT indexed by AccountId
```

Events

An event is a circumstance of interest that occurs within the System or the Monitoring System. By defining such a circumstance as an event definition, it may be reused in other parts of the Valour script. In this way, multiple rules can be created referring to the same event.

An event is composed from the following parts:

1. *Trigger* - which defines the circumstance of interest that will *trigger* a new event.
2. Optional *When* - which defines a guard on whether the *trigger* should in fact generate the event.
3. Optional *Where* - which defines the data to be attributed to the event.
4. Optional *Categorisation* - which defines what categories this event is related to.

An event is thus defined using the following syntax.

```
event <event-name>(<event-parameters>) = {
        <trigger>
        <when>
        <where>
        <categorisation>
}
```

The following sections will describe each in detail, giving practical examples on how these constructs can be used.

Triggers

A *Trigger* defines a circumstance of interest which should be translated into an event. There are different types of events and each will be explained below.

System Control Flow Trigger

The system control flow trigger makes it possible to define a trigger on particular code locations in the system (to be monitored) using an AOP call pointcut expression. Using AOP, we can instrument the system to *trigger* the event creation. Note that using AOP allows the user to create these trigger points in the built system both where source code is available, and also on libraries used by the system whereby source code is not always available.

A system control flow trigger is identified by the `system controlflow trigger` keywords as shown in this template below.

```
system controlflow trigger <AOP Call Pointcut Expression>
```

For the initial version of Valour, the compiler will only support call pointcut expression that will be used to generate advices that will be triggered before a method execution. In the future, it is envisaged that Valour will support other kinds of advices and pointcuts such as after method invocation and around method invocation.

An example of a system control flow trigger is shown below.

```
system controlflow trigger AuthSystem.login(User u)
```

In this example, the Valour compiler will create an advice that will be triggered before the execution of the method `login(User u)` in the `AuthSystem` class, and the advice's implementation will notify the RV Tool in order to generate the associated event.

Kindly refer to the appendix about aspect oriented programming for more information about AOP and how a system control trigger is converted into an advice.

Event Trigger

An event trigger defines the triggering of another event as defines a circumstance of interest i.e. it allows the creation of a new event when another event has already triggered. The syntax for defining an event trigger is shown below.

```
event trigger <event-name>(<event-parameters>)
```

An example of when this trigger can be used is for creating more specific events tied to a particular event. Assume the following event is already defined in the system that notifies the monitors of a new user login.

```
event UserLogin(User u) = {
        system controlflow trigger AuthSystem.login(User u)
        ...
}
```

Then we can define a *GoldUserLogin* event based on the *UserLogin* event defined above as shown below. Note that the example is using the *when* clause which has not be explained as yet. All you need to know at this point is that an event is only triggered if the *when* clause evaluates to true.

```
event GoldUserLogin() = {
        event trigger UserLogin(User u)
        when {{ u.isGoldUser() }}
}
```

Monitor Trigger

A monitor trigger allows monitors to trigger events. In other words, given a monitor trigger a rule can trigger an event as part of its actions (Note: rules and rule actions will be defined later). The example below shows a monitor trigger called forceRestart that triggers a Restart event.

```
event Restart(String reason) = {
        monitor trigger forceRestart(String reason)
        ...
}
```

Timer Trigger

TBD

System Data Flow Trigger

Currently out of scope in this initial version of Valour. As summary, this trigger would be used for monitoring data value changes within the system. For example, when the average number of logins within the last five minutes falls above 1000.

Trigger Groups

Sometimes, a single event may be triggered from different circumstances, and it is desirable to be able to declare the event once and attach it to multiple triggers. For example, a User Login event might require two control flow triggers - the first one to capture desktop logins and the second one to capture mobile device logins.

Valour allows the user to define an event which is generated from a disjunction of triggers, and the triggers may be of any type. The basic structure for creating a disjunction of triggers is shown below.

```
event <event-name>(<event-parameters>) = {
        <trigger-1> || <trigger-2> || <trigger-3> || ... || <trigger-n>
        ...
}
```

An example is shown below, whereby a login event is generated for two control flow triggers (desktop and mobile login).

```
event UserLogin(User u) = {
        system controlflow trigger AuthSystem.login(User u) ||
        system controlflow trigger MobileAuthSystem.login(User u)
        ...
}
```

Note that trigger groups will be mentioned again in the *where* clause section, to describe how a *where* clause can be applied to trigger groups.

When Clause

A Condition defines a guard that decides whether the *trigger* should in fact generate the event or not. A condition is specified in the event definition as a *when* clause, and take the following basic structure.

```
when <condition>
```

Conditions may be defined in three different ways as outlined below.

1. Using the {{ }} notation. Using this notation, the user may define an expression that evaluates to a boolean condition that references variables defined in the trigger expression. An example was already shown for the *GoldUserLogin* defined previously (and shown again below).

```
event GoldUserLogin() = {
        event trigger UserLogin(User u)
        when {{ u.isGoldUser() }}
}
```

2. Using the { } notation. Using this notation, the user may define more complex logic rather than just supplying a simple boolean expression. Expanding on the *GoldUserLogin*, assume that the *User* class does not define *isGoldUser()*, and that a *User* is considered to be a *Gold User* if the User is a resident of Belgium or the *User* has acquired over 10,000 bonus points. Then the *GoldUserLogin* can be re-written to look as follows.

```
event GoldUserLogin() = {
        event trigger UserLogin(User u)
        when {
                boolean isGoldUser = false;
                if (u.getCountryOfResidence().equals("BE"))
                {
                        isGoldUser = true;
                }
                else if (u.getBonusPoints() > 10000)
                {
                        isGoldUser = true;
                }

                return isGoldUser;
        }
}
```

3. Referencing a defined condition. Similar to the way that *Events* are declared to be reused in the *Rules* section, so can *Conditions* be declared and reused. Declaration of conditions will be covered in more detail later on, however the above *GoldUserLogin* may be rewritten as follows.

```
condition isGoldUser(User u) = {
        boolean isGoldUser = false;
        if (u.getCountryOfResidence().equals("BE"))
        {
                isGoldUser = true;
        }
        else if (u.getBonusPoints() > 10000)
        {
                isGoldUser = true;
        }

        return isGoldUser;
}

event GoldUserLogin() = {
        event trigger UserLogin(User u)
        when #isGoldUser(u)
}
```

Where Clause

The Where clause is an optional clause that allows the user to define the parameter values for the event to be extracted from the trigger parameters. By default, Valour will attempt to link parameters based on their name. However, using the Where clause, the Valour script writer has

the capability to rename event parameters, or to compute event parameters from other trigger parameters.

A Where clause is specified in the event definition using the *where* keyword, and takes the following basic structure.

```
where
        <event-parameter-name> = <value-expression>
```

Value expressions may be defined in two different ways as outlined below.

1. Using the {{ }} notation. Using this notation, the user may define an expression that evaluates to a value of the event parameter type.
2. Using the { } notation. Using this notation, the user may define more complex logic rather than just supplying a simple expression.

Expanding on the *GoldUserLogin*, let's add the User and the number of bonus points as parameters to the event. Also, let's assume that the residents of Belgium automatically get an extra 10,000 increase to the actual bonus points on record. Note how the User u variable is not defined in the where clause, as Valour will automatically use the User u variable declared in the event trigger.

```
event GoldUserLogin(User u, int bonusPoints) = {
        event trigger UserLogin(User u)
        when {
                boolean isGoldUser = false;
                if (u.getCountryOfResidence().equals("BE"))
                {
                        isGoldUser = true;
                }
                else if (u.getBonusPoints() > 10000)
                {
                        isGoldUser = true;
                }

                return isGoldUser;
        }
        where
                bonusPoints = {
                        int bonusPoints = u.getBonusPoints();
                        if (u.getCountryOfResidence().equals("BE"))
                        {
                                bonusPoints += 10000;
                        }
                        return bonusPoints;
                }

}
```

The where clause (as mentioned earlier) can also be applied to parts of a trigger group since event parameter computation might need to happen differently for different types of triggers.

Assume a user login event with the system user id as a parameter i.e. `event SystemUserLogin(Integer systemUserId)` needs to be created from the following triggers:

- `system controlflow trigger AuthSystem.login(User u)`
- `system controlflow trigger MobileAuthSystem.login(User u)`
- `system controlflow trigger BackofficeAuthSystem.login(Integer userId)`

Attempting to compute the value systemUserId from the above tiggers is nearly impossible to achieve. However, using the where clause on different parts of the trigger group, setting systemUserId is easily achieved as shown below.

```
event SystemUserLogin(Integer systemUserId) = {
	{
		system controlflow trigger AuthSystem.login(User u)
		system controlflow trigger MobileAuthSystem.login(User u)
	}
	where
		systemUserId = {{u.getId()}}
	||
	{
		system controlflow trigger BackofficeAuthSystem.login(Integer userId)
	}
	where
		systemUserId = {{userId}}

	when {
		...
	}
	...

}
```

Categorisation Clause

The categorisation clause allows the user to define whether the event is associated to any category defined in the Valour script. A categorisation clause is specified in the event definition using the *belonging to* keyword, and takes the following basic structure.

```
belonging to <category-name> with index <value-expression>
```

Apart from referencing the category, the definition expects a value expression that defines the key value to be used to identify and index the category. Similar to the *where* clause, the *with index* value expression can use either the { } or the {{ }} notation. Note that the Valour script compiler will ensure that the type returned by the value expression matches the type defined in the category definition.

As an example, we will extend the *GoldUserLogin* defined above to categorise the event per user.

```
category USER indexed by Integer

event GoldUserLogin(User u, int bonusPoints) = {
```

```
        system controlflow trigger AuthSystem.login(User u)
        event trigger UserLogin(User u)
        when {
                boolean isGoldUser = false;
                if (u.getCountryOfResidence().equals("BE"))
                {
                        isGoldUser = true;
                }
                else if (u.getBonusPoints() > 10000)
                {
                        isGoldUser = true;
                }

                return isGoldUser;
        }
        where
                bonusPoints = {
                        int bonusPoints = u.getBonusPoints();
                        if (u.getCountryOfResidence().equals("BE"))
                        {
                                bonusPoints += 10000;
                        }
                        return bonusPoints;
                }
        belonging to USER with index {{u.getId()}}

}
```

The example states that a *GoldUserLogin* event belongs to the USER category, where the USER category instance may be indentified by the system user id (i.e. `u.getId()`).

Conditions

In the declarations section, the user may define Conditions that can be reused in the Events and the Rule sections.

A condition is defined using the following format.

```
condition <condition-name>(<condition-params>) = <condition-expression>
```

The condition name allows the condition defined to be reused in other parts of the Valour script. A condition may take a number of parameters that will be used to evaluate the condition result. Finally the condition expression may be defined in two ways,

1. Using the {{ }} notation. Using this notation, the user may define an expression that evaluates to a boolean condition. The expression may use a number of parameters in the evaluation process. An example was already shown for the *isGoldUser* condition for the *GoldUserLogin* defined previously will be redefined as a single line expression.

```
condition isGoldUser(User u) = {{ u.isGoldUser() }}
```

2. Using the { } notation. Using this notation, the user may define more complex logic rather than just supplying a simple boolean expression. This logic may use a number of parameters in the evaluation process. An example was already shown for the *isGoldUser* condition for the *GoldUserLogin* defined previously (and shown again below).

```
condition isGoldUser(User u) = {
        boolean isGoldUser = false;
        if (u.getCountryOfResidence().equals("BE"))
        {
                isGoldUser = true;
        }
        else if (u.getBonusPoints() > 10000)
        {
                isGoldUser = true;
        }

        return isGoldUser;
}
```

Actions

In the declarations section, the user may define a number of actions that can be referenced in the Rules section.

An action is defined using the following format.

```
action <action-name>(<action-params>) = <action-logic>
```

The action name allows the action to be reused in other parts of the Valour script. An action may take a number of parameters that will be used during the execution of the action. Finally the action logic is defined within a pair of curly brackets and specifying Java code. Within this block code, the user is allowed to refer to other Valour actions and conditions by pre-pending the action and condition names used the #. Also, from within this block, the user may trigger a *Monitor Trigger* by using the `#generate trigger` keywords.

Some example of actions (including any necessary and related declarations) are shown below.

```
condition isGoldUser(User u) = {{ u.isGoldUser() }}

event goldUserBlockedEvent(User u) = {
        monitor trigger goldUserBlocked(User u)
}


action blockUser(User u) = {
        //block user
        u.blockUser();

        //if user is gold user, then escalate
        if (#isGoldUser(u)) {
```

```
                #generate trigger goldUserBlocked(u);
                 #informCustomerServiceUserIsBlocked(i);
        }

}

action informCustomerServiceUserIsBlocked(User u){
        //send email
}
```

Rules

The Rules section allows the user to define the rules for monitoring the system. As explained in the introduction, the basic form of a rule is: On **Event**, If **Condition**, Then **Action**. Rules may be defined in two different locations within the Valour script:

1. At the top most level of the script right after the *declarations* sections.
2. Embedded within constructs that maintain monitoring state.

This section will first introduce the syntax for defining a basic rule that exists without any monitoring state, and later defines the various ways in which state may be defined and how rules can be attached to the given state.

Basic Rule

A basic rule is triggered whenever an event is observed, and if the specified condition evaluates to true then the associated rule actions are executed. The following is the syntax for defining a basic rule.

```
<event> | <condition> -> <action>
```

Event

The event part of the rule references an event that was declared in the declarations section of the Valour script. The event in the rule will need to also define any parameters that are defined in the event. In this way, these parameters are usable within the conditions and the actions clauses.

Condition

Once an event is observed and a rule triggers, the condition acts as a guard for the execution of the event actions. As for the condition definition in an event declaration, a condition in a rule may be defined in one of the following ways

1. Using the {{ }} notation. Using this notation, the user may define an expression that evaluates to a boolean condition that references variables defined in the event parameters or in the associate state (to be defined later).

2. Using the { } notation. Using this notation, the user may define more complex logic rather than just supplying a simple boolean expression. This logic may reference variables defined in the event parameters or in the associated state.
3. Referencing a defined condition.

Action

If the rule condition evaluates to true, then the rule actions are executed. Actions to be executed may be defined in one of the following ways:

1. Using the { } notation as described in the action declaration.
2. Referencing a defined action.

Example

```
declarations {

	condition isGoldUser(User u) = {{ u.isGoldUser() }}

	condition shouldBlockUser(User u) = {{ u.getConsecutiveFailedLoginCount() >= 5}}

	event failedLogin(User u) = {
		system controlflow trigger LoginHandler.failedLogin(User u)
	}


	action blockUser(User u) = {
		//block user
		u.blockUser();

		//if user is gold user, then escalate
		if (#isGoldUser(u)) {
			#informCustomerServiceUserIsBlocked(i);
		}
	}

	action informCustomerServiceUserIsBlocked(User u){
		//send email
	}
}

failedLogin(u) | shouldBlockUser(u) -> blockUser(u)
```

The above example shows the construction of a basic rule using events, conditions and actions defined in the declarations section. Let's decompose the rule:

- Event: The basic rule is attaching itself to the *failedLogin(u)* event whereby it is being notified of a failed login for user *u*.
- Condition: The rule defines a condition guarding the rule action execution - it restricts the action to be run only if the user should be blocked (i.e. if this is the 5th consecutive login failure). Note that the condition has already been defined in the declarations section, and is merely referenced in the rule definition.

- Action: The action to be executed references an action already defined in the declarations section, which in this case blocks the user.

To summarise, the rule is stating *"On a user login failure, if this is at least the fifth failed login, then block the user."*

Defining State

The Basic Rule presented above is stateless. It is desirable, that rules maintain their own internal state that can be used during rule condition evaluation or rule action execution. Valour allows the user to create this state and to attach rules that use this state. Apart from allowing the state to be used in the rule definitions, Valour also allows the user to define new declarations that make use of the state. In other words, the state may be defined for a whole new Valour script (excluding imports).

The basic syntax for doing this is presented below.

```
state {
        <state-declarations>
}
in {
        <declarations-block>?
        <rule>*
}
```

Firstly, the user starts off the first block with the *state* keyword and declares any state to be maintained. State declarations are normal Java variable declarations with optional state initialisation. State may be initialised in one of the following ways:

1. Using Java Defaults. If left uninitialised, the declaration will be initialised with the Java default initialisation for that type. Kindly refer to the Java language specification for the default initialisation values (Java 7, Java 8).
2. Using the { } notation described previously, which accepts a Java value expression.
3. Using the {{ }} notation described previously, which accepts a Java code block with a return statement returning the initialisation value.

Note that apart from declaring variables that hold state for the monitoring system, this section may also be used to initialise handles to system services and resources that need to be used in the actions, conditions and rules.

In the next block marked with the *in* keyword, the user will define the rules that can make use of the declared state. The following example shows how state can be defined for rules by building on the example presented for the Basic Rule.

```
import com.example.User;

declarations {
```

```
	condition isGoldUser(User u) = {{ u.isGoldUser() }}

	condition shouldBlockUser(User u) = {{ u.getConsecutiveFailedLoginCount() >= 5}}

	event failedLogin(User u) = {
		system controlflow trigger LoginHandler.failedLogin(User u)
	}

	event userBlocked(User u) = {
		monitor trigger userBlockedTrigger(User u)
	}

	action blockUser(User u) = {
		//block user
		u.blockUser();


		//if user is gold user, then escalate
		if (#isGoldUser(u)) {
			#informCustomerServiceUserIsBlocked(i);
			#generate trigger userBlockedTrigger(u);
		}
	}

	action informCustomerServiceUserIsBlocked(User u){
		//send email
	}

}

state {
	Long totalFailedLogins = {{ 0L }}
	Boolean tooManyBlockedUsers = {{ Boolean.FALSE }}
	Long totalBlockedUsers = { return 0L; }
}
in {
	declarations {
		action incrementTotalFailedLogins() { totalFailedLogins++; }
		action incrementTotalBlockedUsers() { totalBlockedUsers++; }
	}

	failedLogin(u) -> { #incrementTotalFailedLogins; }
	userBlocked(u) -> { #incrementTotalBlockedUsers; }
	userBlocked(u)
		| {{ totalBlockedUsers > 1000 }}
		-> { tooManyBlockedUsers = Boolean.TRUE; }
}

failedLogin(u) | shouldBlockUser(u) -> blockUser(u)
```

Of note in the above example are the following items:

- The new state block defining the `totalFailedLogins`, `tooManyBlockedUsers`, and `totalBlockedUsers` variables. Note how the former is initialised using the {{ }} notation while the latter is initialised using the { } notation.
- The new state block contains new rules that make use of the state variables, both in the rule condition and the rule action.

- The "`failedLogin(u) | shouldBlockUser(u) -> blockUser(u)`" rule sits outside the state block as it does not utilise any of the state.
- The `incrementTotalBlockedUsers()` and `incrementTotalBlockedUsers()` actions are defined as scoped actions within the state-in block. In this way these actions
    - Are scoped to within the state-in block i.e. can only be referenced within this state-in block.
    - May access any state defined in this state-in block (and anything defined at higher levels).

Replicating State

The capability of defining state, allows the user to create more complex rules. However, it is also desirable to define this state in a scoped (i.e. per category) and replicated fashion. In other words, the Valour script writer might want to create a replicated state for each particular domain object. For example, the script writer might be interested in maintaining the number of failed logins per user in order to define a rule that states: "On Failed Login, If this is the fifth consecutive failed login, Then user should be blocked. Or, for example, the script writer might be interested in storing creation date of an HTTP session and the number of requests received over the session in order to define a rule that states: "On any request, If the request rate is greater than 500 requests/second, Then the session should be destroyed". Maintaining state per given domain objects, allows the script writer to reason about the system in a more granular fashion.

The key to replicating state is to define the category for which state will be replicated. So far we've seen how a Category is declared, and how Events are marked as belonging to a Category. This section describes how state may be replicated for each Category.

The basic syntax for defining state to be replicated per category is presented below. The structure can be read as follows "Replicate the declared state for every category instance encountered while processing the below rules set."

```
replicate {
        <state-declarations>
}
foreach <category> <category-label> {
        <declarations-block>?
        <rule>*
}
```

The *replicate* block is similar to the *state* block presented in the previous section, while the *foreach* block is similar to the *in* block. Note that similarly to the *in* block, the *foreach* block may define a whole new Valour script (excluding imports).

The following Valour script extends the example presented in the State section so that the counters are also held per user instead of across all the system.

```
import com.example.User;

declarations {
```

```
        category USER indexed by Integer

        condition isGoldUser(User u) = {{ u.isGoldUser() }}

        condition shouldBlockUser(User u) = {{ u.getConsecutiveFailedLoginCount() >= 5}}

        event failedLogin(User u) = {
                system controlflow trigger LoginHandler.failedLogin(User u)
                belonging to USER with index {{u.getId()}}
        }

        event userBlocked(User u) = {
                monitor trigger userBlockedTrigger(User u)
                belonging to USER with index {{u.getId()}}
        }

        action blockUser(User u) = {
                //block user
                u.blockUser();


                //if user is gold user, then escalate
                if (#isGoldUser(u)) {
                        #informCustomerServiceUserIsBlocked(i);
                        #generate trigger userBlockedTrigger(u);
                }
        }

        action informCustomerServiceUserIsBlocked(User u){
                //send email
        }

}

state {
        Long totalFailedLogins = {{ 0L }}
        Boolean tooManyBlockedUsers = {{ Boolean.FALSE }}
        Long totalBlockedUsers = { return 0L; }
}
in {
        declarations {
                action incrementTotalFailedLogins() { totalFailedLogins++; }
                action incrementTotalBlockedUsers() { totalBlockedUsers++; }
        }

        userBlocked(u)
                | {{ totalBlockedUsers > 1000 }}
                -> { tooManyBlockedUsers = Boolean.TRUE; }

        replicate {
                Long userFailedLogins = {{ 0L }}
                Boolean userBlocked = {{ Boolean.FALSE }}
        }
        foreach USER u {
                failedLogin(u) ->
                                {
                                        #incrementTotalFailedLogins();
                                        userFailedLogins++;
                                }
                userBlocked(u) ->
                                {
                                        #incrementTotalBlockedUsers();
                                        userBlocked = Boolean.TRUE;
                                }
```

```
        }
}

failedLogin(u) | shouldBlockUser(u) -> blockUser(u)
```

Of note in the above example are the following items:

- The new replicate block defining the userFailedLogins and total userBlocked variables.
- The new replicate-foreach block containing new rules that make use of the foreach state variables.
- The replicate-foreach block is nested within the state block. This allows the replicate-foreach rules to use state defined above. Nesting will be discussed in more detail later.

Parallel State Replication

Replicate blocks allow the replication of a given state per Category, thus providing an easy way for maintaining complex state about the system. However, the RV Tool will process events related to a replicate block in a sequential fashion in order to to avoid concurrent modification and access of shared data. For instance, using the example defined above, let's assume that User A and User B log into the system in quick succession. In this case, the RV Tool will first process the rules defined in the replicate block associated to the login for User A and once completed, the RV Tool will then process the rules associated to the login event for User B.

However, it is desirable that independent events that do not share any data are processed concurrently. The *replicate-in-parallel* block allows this to happen while shifting the responsibility of concurrent data access safety to the script writer. Note that this does not only apply to state maintained within the monitoring system, but also applies to any state maintained within the system. In other words, if actions and conditions are accessing some state that is stored on the system side, then it is the responsibility of the system developer to ensure that this data access happens safely.

The basic syntax for defining state to be replicated in parallel is presented below.

```
replicate in parallel {
        <state-declarations>
}
foreach <category> <category-label> {
        <declarations-block>?
        <rule>*
}
```

The following Valour script extends the example presented in the Replicate State section so that the foreach events are processed in parallel.

```
import com.example.User;

declarations {
```

```
        category USER indexed by Integer

        condition isGoldUser(User u) = {{ u.isGoldUser() }}

        condition shouldBlockUser(User u) = {{ u.getConsecutiveFailedLoginCount() >= 5}}

        event failedLogin(User u) = {
                system controlflow trigger LoginHandler.failedLogin(User u)
                belonging to USER with index {{u.getId()}}
        }

        event userBlocked(User u) = {
                monitor trigger userBlockedTrigger(User u)
                belonging to USER with index {{u.getId()}}
        }

        action blockUser(User u) = {
                //block user
                u.blockUser();


                //if user is gold user, then escalate
                if (#isGoldUser(u)) {
                        #informCustomerServiceUserIsBlocked(i);
                        #generate trigger userBlockedTrigger(u);
                }
        }

        action informCustomerServiceUserIsBlocked(User u){
                //send email
        }

}

state {
        AtomicLong totalFailedLogins = {{ new AtomicLong() }}
        AtomicBoolean tooManyBlockedUsers = {{ new AtomicBoolean() }}
        AtomicLong totalBlockedUsers = { return new AtomicLong(); }
}
in {
        declarations {
                action incrementTotalFailedLogins() {
                        totalFailedLogins.incrementAndGet();
                }
                action incrementTotalBlockedUsers() {
                        totalBlockedUsers.incrementAndGet();
                }
        }

        userBlocked(u)
                | {{ totalBlockedUsers.get() > 1000L }}
                -> { tooManyBlockedUsers.getAndSet(Boolean.TRUE); }

        replicate in parallel {
                Long userFailedLogins = {{ 0L }}
                Boolean userBlocked = {{ Boolean.FALSE }}
        }
        foreach USER u {
                failedLogin(u) ->
                                {
                                        #incrementTotalFailedLogins();
                                        userFailedLogins++;
                                }
                userBlocked(u) ->
```

```
                                {
                                        #incrementTotalBlockedUsers();
                                        userBlocked = Boolean.TRUE;
                                }
        }
}

failedLogin(u) | shouldBlockUser(u) -> blockUser(u)
```

Of note in the above example are the following items:

- The `totalFailedLogins`, `tooManyBlockedUsers` and `totalBlockedUsers` have now become AtomicLong and AtomicBoolean respectively. The reason for this is that the rules within the replicate in parallel block can potentially access these variables in a concurrent fashion.
- The `userFailedLogins` and `userBlocked` variables have not become their respective Atomic* counterparts as a dedicated instance will be created for each USER. The RV Tool will guarantee that if multiple concurrent failed logins for USER u are received, they will be processed sequentially.

Nesting State

Valour allows for the state maintaining constructs presented above to be nested within each other. As seen in the examples above, *replicate-foreach* and a *replicate-in-parallel* blocks have been nested in a *state* block. When this occurs, Valour applies scoping rules to the variables declared in the state blocks, picking up the closest matching variable first.

The following example shows how state and rules can be nested, and the comments show how the variable will be evaluated depending on their scope.

```
import com.example.User;

declarations {

        category USER indexed by Integer

        event userLogin(User u) = {
                system controlflow trigger LoginHandler.userLogin(User u)
                belonging to USER with index {{u.getId()}}
        }
}

state {
        String x = {{ “X - Level 0” }}
        String y = {{ “Y - Level 0” }}
}
in {
        replicate {
                String x = {{ “X - Level 1” }}
        }
        foreach USER u {
                userLogin(u) -> {
                        System.out.println(x); //will print “X - Level 1”
                        System.out.println(y); //will print “Y - Level 0”
                }
```

```
        }

        userLogin(u) -> {
                System.out.println(x); //will print “X - Level 0”
                System.out.println(y); //will print “Y - Level 0”
        }
}
```

Conclusion

This document presents Valour, a language designed for specifying and generating a monitoring system for a given system. The language provides different mechanisms for defining events - circumstances of interest happening within the system (and the monitoring system). Given these event definitions, monitoring rules can be defined that define what actions need to be executed when an event occurs and given certain conditions are satisfied. The language also provides constructs that allow the creation of state and replicated state for specific domain objects.

As this is the first version of Valour, it is envisage that new features and other adaptations will be introduced as its usage picks up. In particular, it is noteworthy to mention the expansion of the system controlflow trigger to other types of AOP expressions, and to provide new types of triggers such as timer based triggers and data flow triggers.

Appendix A: BNF

```
<valour> ::= [ <imports> ] <valour-body>;

<imports> ::= { "import" <fully-qualified-java-class> ";" };

<valour-body> ::= [ <declarations> ] { <rule> };

<declarations> :: = "declarations" "{" { <declaration> } "}";

<declaration> ::= <category-dec> | <event-dec> | <condition-dec> | <action-dec>;

<category-dec> ::= "category" <identifier> "indexed by" <java-type>;

<event-dec> ::= "event" <identifier>"(" [<formal-parameters] ")" "=" "{" <event-body>
"}";

<formal-parameters> ::= <formal-parameter> { "," <formal-parameter> };

<formal-parameter> ::= <java-type> <identifier>;

<event-body> ::= <trigger> [ <when-clause> ] [ <where-clause> ] [
<categorisation-clause> ];

<trigger> ::= <system-controlflow-trigger> | <event-trigger> | <monitor-trigger> |
<trigger-group>;

<system-controlflow-trigger> ::= "system controlflow trigger" <aop-expression>;

<event-trigger> ::= "event trigger" <identifier> "(" [<formal-parameters>] ")";

<monitor-trigger> ::= "monitor trigger" <identifier> "(" [<formal-parameters>] ")";

<trigger-group> ::= <trigger> [<where-clause>] { "||" <trigger> [<where-clause>] };

<when-clause> ::= "when" ( <condition-ref> | <condition> );

<condition-ref> ::= "#"<label>"(" [ <actual-parameters> ] ")";

<actual-parameters> ::= <actual-parameter> { "," <actual-parameter> };

<actual-parameter> ::= <java-expression>;

<condition> ::= <simple-condition> | <complex-condition>;

<simple-condition> ::= "{{" <boolean-expression> "}}";

<complex-condition> ::= "{" <java-boolean-method-body> "}";

<where-clause> ::= "where" <identifier> "=" <value-expression> {<identifier> "="
<value-expression>};

<value-expression> ::= <simple-value-expression> | <complex-value-expression>;
```

```
<simple-value-expression> ::= "{{" <java-expression>"}}";

<complex-value-expression> ::= <java-method-body>;

<categorisation-clause> ::= "belonging to" <identifier>"with index" <value-expression>;

<condition-dec> ::= "condition" <identifier>"(" [ <formal-parameters> ] ")" =
<condition>;

<action-dec> ::= "action" <identifier>(" [ <formal-parameters> ] ")" = <action>;

<action> ::= "{" <action-body> "}";

<action-body> ::= <java-void-method-body> | <action-ref> | <monitor-trigger-fire>;

<action-ref> ::= "#"<identifier> "(" [ <actual-parameters> ] ")";

<monitor-trigger-fire> ::= "monitor trigger" <identifier>"(" [ <actual-parameters> ]
")";

<rule> ::= <basic-rule> | <state-block> | <foreach> | <par-foreach>;

<basic-rule> ::= <event-ref> [ "|" (<condition-ref> | <condition>) ] "->" <action>;

<event-ref> ::=<identifier> "(" [ <actual-parameters> ] ")";

<state-block> ::= "state" "{" <state-declarations> "}" "in" "{" [<valour-body>] "}" ;

<state-declarations> ::= { <state-declaration> };

<state-declaration> ::= <java-type> <identifier> [ "=" <value-expression> ];

<foreach> ::= "replicate" "{" <state-declarations> "}" "foreach" "{" [<valour-body>] "}"
;

<par-foreach> ::= "replicate in parallel" "{" <state-declarations> "}" "foreach" "{"
[<valour-body>] "}" ;
```

Appendix B: Aspect Oriented Programming

What is Aspect Oriented Programming?

Aspect Oriented Programming (AOP) is a programming paradigm with the aim of tackling crosscutting concerns such as security, audit logging, transaction management, validation etc. AOP aims to modularise the way in which crosscutting concerns are developed in order to add functionality without requiring any changes to the existing code base.

There are four main concepts that need to be described in order to explain the AOP paradigm.

- **Join point** - Join points are specific points at which a crosscutting concern may join the code in order to execute its specific logic. Examples of such join points are method join points that join the code when a method is invoked; or field accessor join points that join the code when fields are being read or written.
- **Pointcut** - A pointcut is a definition of a specific join point. A class FooBar with two methods(foo() and bar()) has two *method invocation* joint points, and a *method invocation* pointcut will define the methods (one or more) for which the crosscutting concern logic will join and execute.
- **Advice** - An advice defines how interaction with a point occurs (before/after/around) and what will happen at that interaction (i.e. logic to execute). For example, a method entry logging advice for FooBar would be defined as a *before* method invocation pointcut on methods foo() and bar().
- **Aspect** - Aspects provide a modular unit of declaring pointcuts and advices for a single crosscutting implementation.

AspectJ

The most popular implementation of the AOP paradigm in Java is AspectJ, and the RV Tool and the Valour compiler will be using AspectJ in order to join the system code and generate the required events.

The following is a logging aspect written in AspectJ that will join the methods of the class FooBar before their invocation in order to log method entry.

```
aspect Logging {
	before() : call (void FooBar.foo()) {
		System.out.println("Before foo");
	}

	before() : call (void FooBar.bar()) {
		System.out.println("Before bar");
	}
}
```

AOP in Valour and the RV Tool

As mentioned, the Valour compiler and the RV Tool will be using AspectJ in order to join the system code and generate the required event for the system control flow triggers. A system

control flow trigger is identified by the system controlflow trigger keywords as shown in this template below.

```
system controlflow trigger <AOP Call Pointcut Expression>
```

The <AOP Call Pointcut Expression> will be used to create a before method invocation advice. In other words, the Valour script accepts a method call pointcut expression and then places it in the following template.

```
before() : call (<AOP Call Pointcut Expression>) {
        ...
}
```

For example, a system controlflow trigger on FooBar.foo() would look like this in Valour.

```
system controlflow trigger FooBar.foo()
```

The Valour compiler would then convert the system control flow trigger into the following advice.

```
before() : call (FooBar.foo()) {
        ...
}
```

Future Work

So far, Valour only supports an AOP call pointcut expression as part of the system control flow trigger. It is envisaged that future versions of Valour (and the RV Tool) will support a more rich AOP expression allowing the Valour script writer to attach to the system code at other joint points. Two such examples include:

- The capability of defining an after method call execution pointcut that would supply in the event data the return value or the Exception thrown after method execution.
- The capability of defining an around method call execution pointcut that would allow rules to control the execution flow of the system i.e. whether to continue with the method execution or whether to abort.