YAWL 4.1 example description

This document describes an example demonstrating process modelling capabilities supported by YAWL (version 4.1). The example describes a process followed by engineers in a processing centre to configure a new terminal (ATM or POS) for a bank ('Add new terminal' process).

Specification files

Category	File	Description
Main YAWL specification	*Add_Terminal.yawl*	Top-level process specification
Organisational model	OrgData_Add_terminal.ybkp	Organisational data (password for all users: YAWL)
Cost model	AddTerminalCostModel.xml	Cost model for the process
Worklets	Fix_Major_Issue.yawl Fix_Minor_Issue.yawl Fix_Locally.yawl Handle_TestFailed.yawl HandleUrgentMajorIssue.yawl Recall.yawl	Worklet specifications
Rule sets	Add_Terminal.xrs Fix_Locally.xrs Fix_Major_Issue.xrs Fix_Minor_Issue.xrs	Rule sets used in the example

Contents

Process description

Control-flow perspective

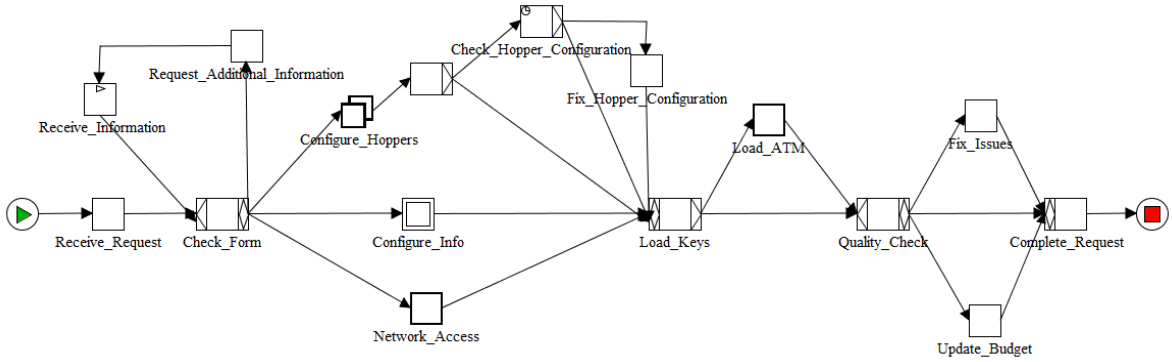

Figure 1: Main steps of the 'Add new terminal' process.

Figure 1 specifies main steps followed by engineers in a processing centre to add information about new terminals (*ATM* or *POS*) on the host. The process starts when a bank lodges a request for adding a new terminal on the host (task *Receive_Request*). Then engineer checks information provided by the bank (task *Check_Form*) and requests additional information if it is needed (task *Request_Additional_Information*). Once all information is received, it is added on the host (composite task *Configure_Info*). For ATM terminals network access should be allowed (task *Network_Access*) and ATM hoppers should be configured (multiple instance task *Configure_Hoppers*). An ATM terminal can use from one to four hoppers; hence, 1-4 instances of the *Configure_Hoppers* task are started during the execution. If the terminal installation is not urgent, the hoppers configuration is double-checked by another engineer (task *Check_Hopper_Configuration*) and fixed if necessary (task *Fix_Hopper_Configuration*). Task *Check_Hopper_Configuration* has an associated timer: the task completion will be forced if it is not completed within a given time frame. Then security keys are loaded on the host to enable safe communication with the terminal (task *Load_Keys*). ATM terminals should be started after this (task *Load_ATM*). For POS terminals tasks *Network_Access*, *Configure_Hoppers* and *Load_ATM* are not performed. Then an engineer checks the quality of the terminal installation (task *Quality_Check*). At this point in the process the overall cost of the case is checked and if the cost exceeds a threshold, the case budget is updated (task *Update_Budget*). If there are no quality issues, an engineer fills in a form with the host specification information for the terminal which is sent to the bank (task *Complete_Request*). If there is a problem with the terminal installation, task *Fix_Issues* is performed.

Data perspective

Figure 2 depicts simple and complex data types defined for the process.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:simpleType name="Terminal">
  <xs:restriction base="xs:string">
   <xs:enumeration value="POS" />
   <xs:enumeration value="ATM" />
  </xs:restriction>
 </xs:simpleType>
 <xs:simpleType name="Urgency">
  <xs:restriction base="xs:string">
   <xs:enumeration value="Urgent" />
   <xs:enumeration value="Normal" />
  </xs:restriction>
 </xs:simpleType>
 <xs:simpleType name="Quality">
  <xs:restriction base="xs:string">
   <xs:enumeration value="No_Issues" />
   <xs:enumeration value="Minor_Issue" />
   <xs:enumeration value="Major_Issue" />
  </xs:restriction>
 </xs:simpleType>
 <xs:simpleType name="Currency">
  <xs:restriction base="xs:string">
   <xs:enumeration value="AUD" />
   <xs:enumeration value="USD" />
   <xs:enumeration value="EURO" />
  </xs:restriction>
 </xs:simpleType>
 <xs:complexType name="Hopper">
  <xs:sequence>
   <xs:element name="Currency" type="Currency" />
   <xs:element name="Amount" type="xs:integer" default="0" />
  </xs:sequence>
 </xs:complexType>
 <xs:complexType name="Hoppers">
  <xs:sequence>
   <xs:element minOccurs="0" maxOccurs="4" name="Hopper" type="Hopper" />
  </xs:sequence>
 </xs:complexType>
</xs:schema>
```

Figure 2: data type definitions

Four simple types were defined (*Terminal*, *Urgency*, *Quality* and *Currency),* each restricting its values to a predefined set, e.g., type *Terminal* allows two possible values: *ATM* or *POS*. Complex type *Hopper* is defined by its *currency* and *amount*, and complex type *Hoppers* is a sequence of maximum 4 elements of type *Hopper*. When a request is lodged, a customer specifies terminal type (*ATM* or *POS*), case urgency (*Normal* or *Urgent*) along with other details about the terminal, e.g. its location and model. For ATM terminal the customer must also define from one to four hoppers by specifying their currencies and amounts. When an engineer checks the quality of configuration in task *Check_Quality*, s/he selects one of the three possible quality values: *No_Issues*, *Major_Issue* or *Minor_Issue*. Variables of types *Terminal*, *Urgency* and *Quality* are used to specify routing conditions.

For example, as depicted in Figure 3, tasks *Network_Access* and *Configure_Hoppers* are only performed when *terminal* = '*ATM*'.

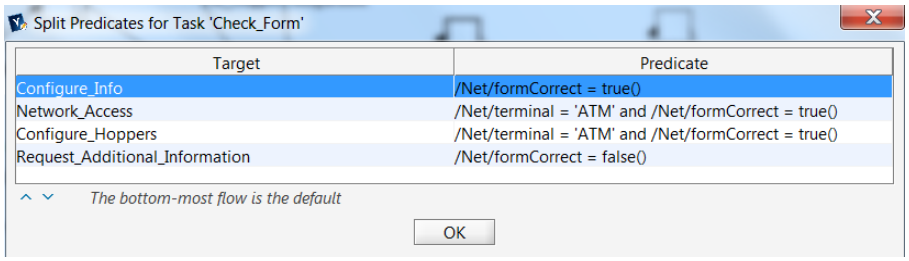

Figure 3: Split Predicates for Task Check_Form

Variable of type *Hoppers* is used by the multiple instance task *Configure_Hoppers* to create multiple (1-4) instances of the task (one for each ATM hopper).

Resource perspective

There are three roles defined for the process: *Engineer*, *Senior Engineer* and *Network Engineer*. Most tasks in the process can be performed by *Engineers*; task *Network_Access* is performed by a *Network Engineer*, and task *Load_Keys* can only be performed by a *Senior Engineer*. Tasks *Request_Additional_Information* and *Complete_Request* must be performed by the same resource who completed task *Receive_Request* (e.g., as depicted in Figure 4) in order to provide a customer a single point of contact. Task *Check_Hopper_Configuration* must be performed by a different resource who completed task *Configure_Hoppers* (as depicted in Figure 5) - "4-eyes principle".

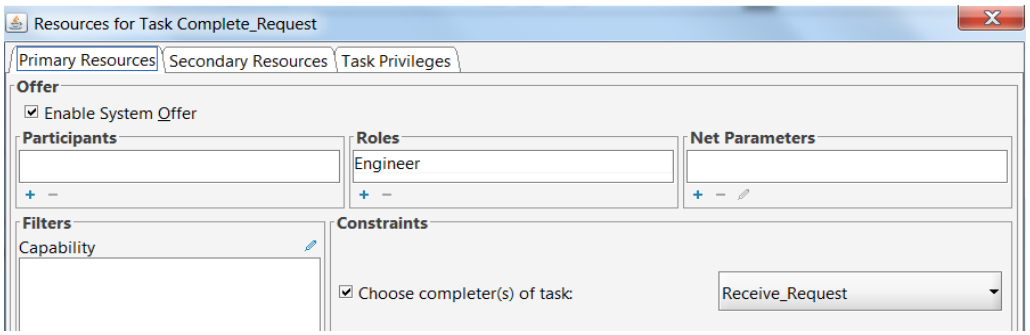

Figure 4: Resources for task 'Complete_Request'

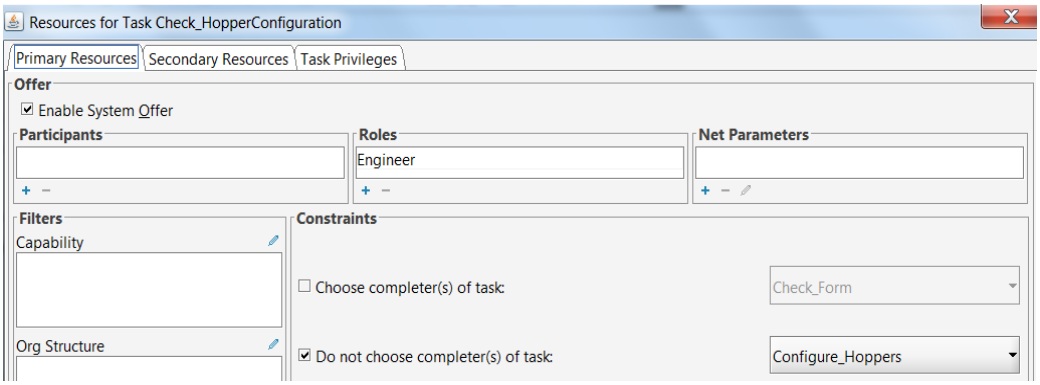

Figure 5: Resources for task 'Check_HopperConfiguration'

Advanced YAWL features

Process configuration

The process can handle two types of terminals: ATM and POS terminals. Most of the tasks are performed for both ATM and POS terminals, except for the tasks *Network_Access*, *Load_ATM*, *Configure_Hoppers* (and consequently tasks *Check_Hopper_Configuration* and *Fix_Hopper_Configuration*), which are only performed for ATM terminals. The process model was configured to reflect these two process variants. Input ports for tasks *Configure_Hoppers*, *Network_Access* and *Load_ATM* are blocked (e.g., as depicted in Figure 6).

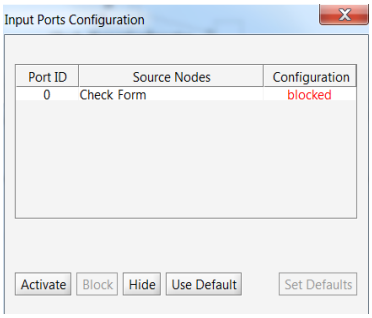

Figure 6: Input Ports Configuration for task 'Network_Access'.

Figure 7 depicts the resulting process model in the 'Preview Process Configuration' mode. Those parts of the process that are common for ATM and POS terminals are depicted in black, while parts of the process that are only executed for ATM terminals are depicted in grey.

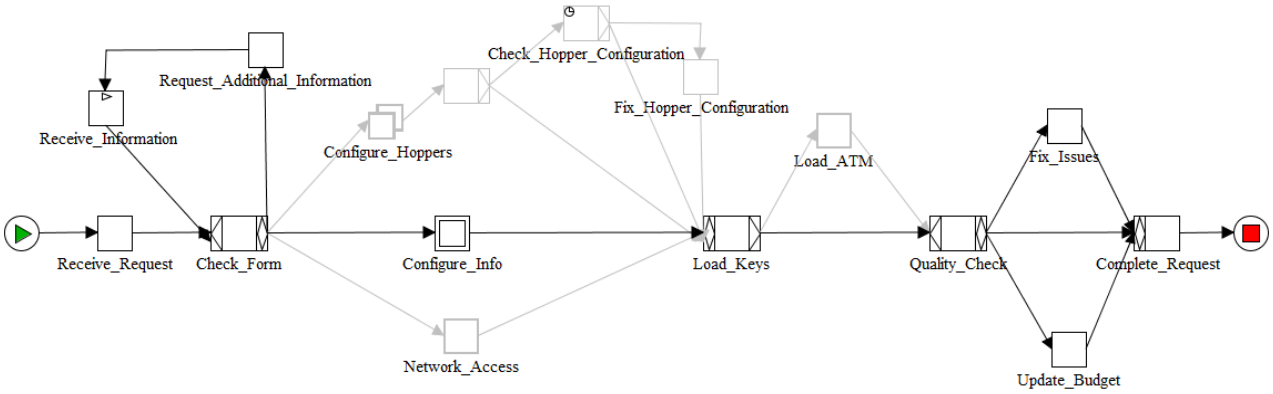

Figure 7: Process Configuration Preview

Worklet selection

Task *Fix_Issues* is associated with the worklet selection service which selects one of the two worklets depending on the value of the variable *Quality*. The worklet selection rule is depicted in Figure 8:

IF Quality= Minor_Issue THEN select Fix_Minor_Issue ELSE IF Quality=Major_Issue THEN select Fix_Major_Issue

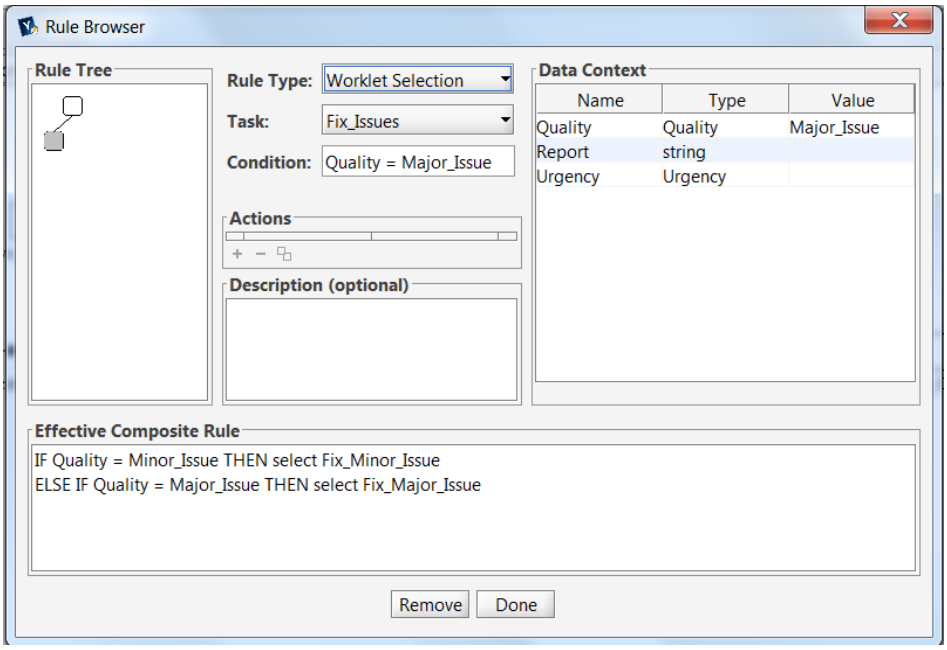

Figure 8: Worklet selection rule for task Fix_Issues

Figures 9 and 10 depict specifications for the *Fix_Minor_Issue* and *Fix_Major_Issue* worklets respectively.

The *Fix_Minor_Issue* process starts with the *Check_Issue* task which identifies the type of the issue. Depending on the issue type, one of the three tasks is performed: *Fix_Terminal_Settings*, *Fix_Host_Setting*, or *Fix_Network_Settings*. Finally, the solution is tested, and the process either completes (if the issue is fixed) or is repeated (if the issue is not fixed).

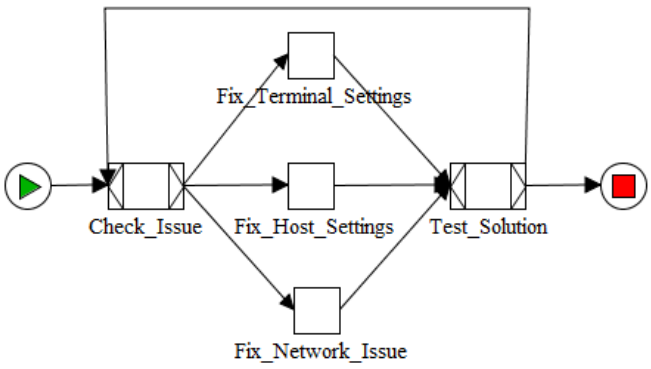

Figure 9: 'Fix_Minor_Issue' worklet specification

The Fix_Major_Issue process starts with the task *Log_Issue_Information* followed by the task *Diagnose_Issues*. If a root cause is identified, then task *Fix_Locally* is performed, otherwise a request is sent to the terminal vendor (task *Forward_to_Vendor*) and in parallel a local investigation is started (task *Local_Investigation*). If task *Local_Investigation* is completed before an answer is received from the vendor, then the vendor request is cancelled; and if the vendor's solution is received earlier, then task *Local_Investigation* is cancelled. For example, the cancellation set for the task *Local_Investigation* is highlighted with red in Figure 10. Finally, the solution is tested, and the process either completes (if the issue is fixed) or is repeated (if the issue is not fixed).

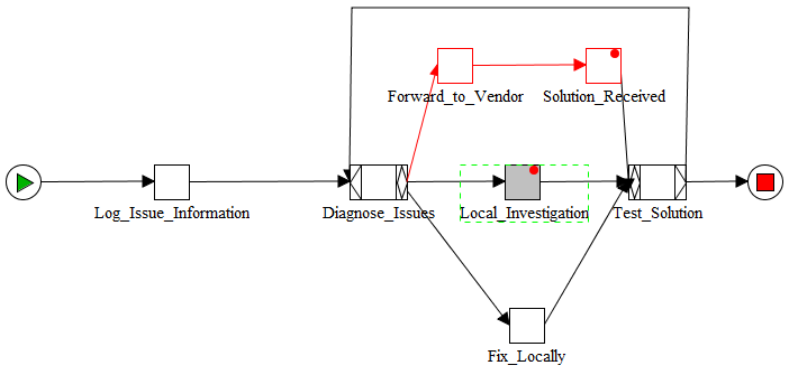

Figure 10: 'Fix_Major_Issue' worklet specification

Exception handling

There are several exceptions defined for the process.

1) "**A major issue in an urgent case**". If a major issue is detected (*Quality=Major_Issue*) and a case is urgent (*Urgency=Urgent*), the case needs to be suspended and additional steps have to be performed: the case completion time has to be updated and the customer has to be notified about the issue. To handle this situation, an exception rule (pre-case constraint violation) is defined in the *Fix_Major_Issue* worklet as depicted in Figure 11. Figure 12 depicts *HandleMajorUrgentIssue* worklet.

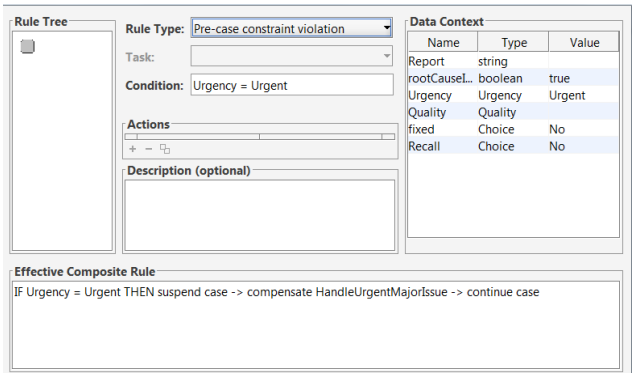

Figure 11: Exception rule definition for an urgent case with a major issue.

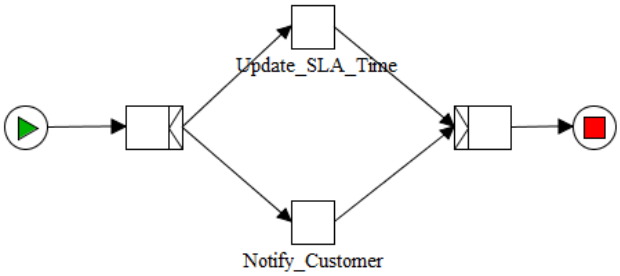

Figure 12: 'HandleMajorUrgentIssue' worklet specification.

2) "**A major issue is not fixed**". Another exception rule defined in the *Fix_Major_Issue* worklet is depicted in Figure 13. If after the execution of task *Test_Solution* (workitem post-constraint violation) the issue is still not fixed, then the case is suspended, *Handle_TestFailed* worklet is executed (depicted in Figure 14), and the case is continued.

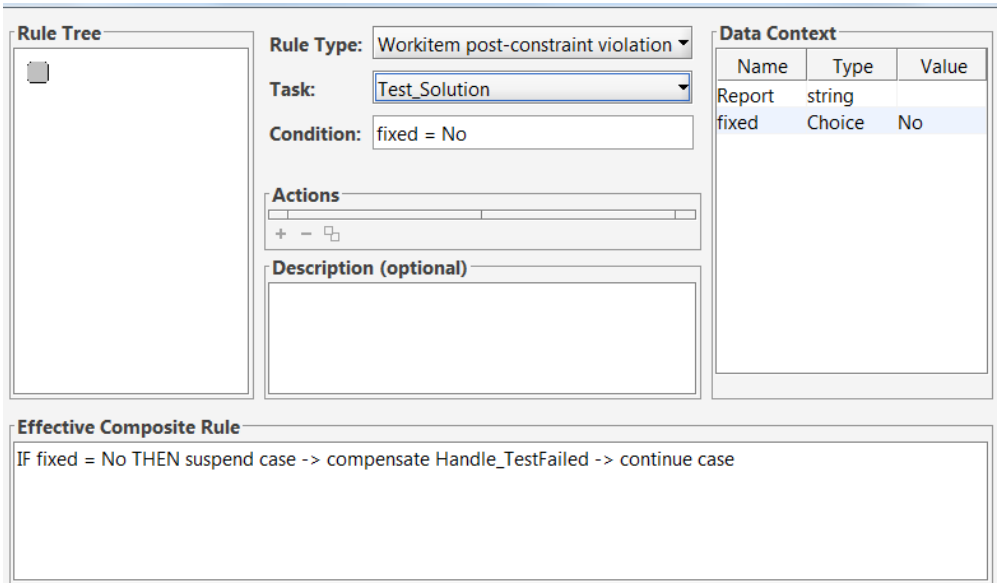

Figure 13: Exception rule definition for the case when a major issue is not fixed.

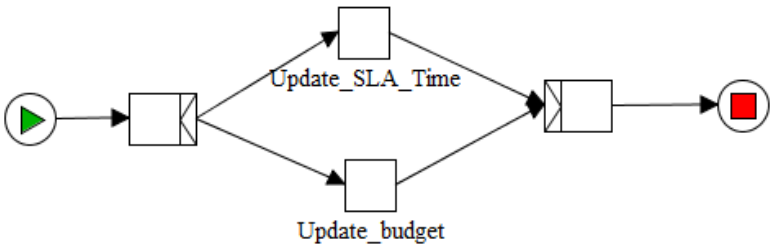

Figure 14: 'Handle_TestFailed' specification.

3) “**Restart a task after a timeout**”. In the *Fix_Major_Issue* process, task *Fix_Locally* is associated with the worklet service and is substituted during the execution with the worklet depicted in Figure 15.

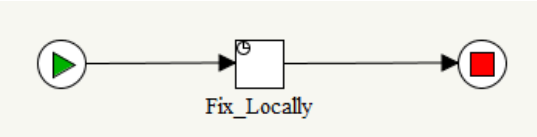

Figure 15: ‘Fix_Locally’ worklet specification.

The only task in the specification has a timer whose expiration triggers a sequence of events depicted in Figure 16: the case is suspended, a new instance of the worklet is started, and the case is continued. This allows restarting the *Fix_Locally* task after the expiration of its timer.

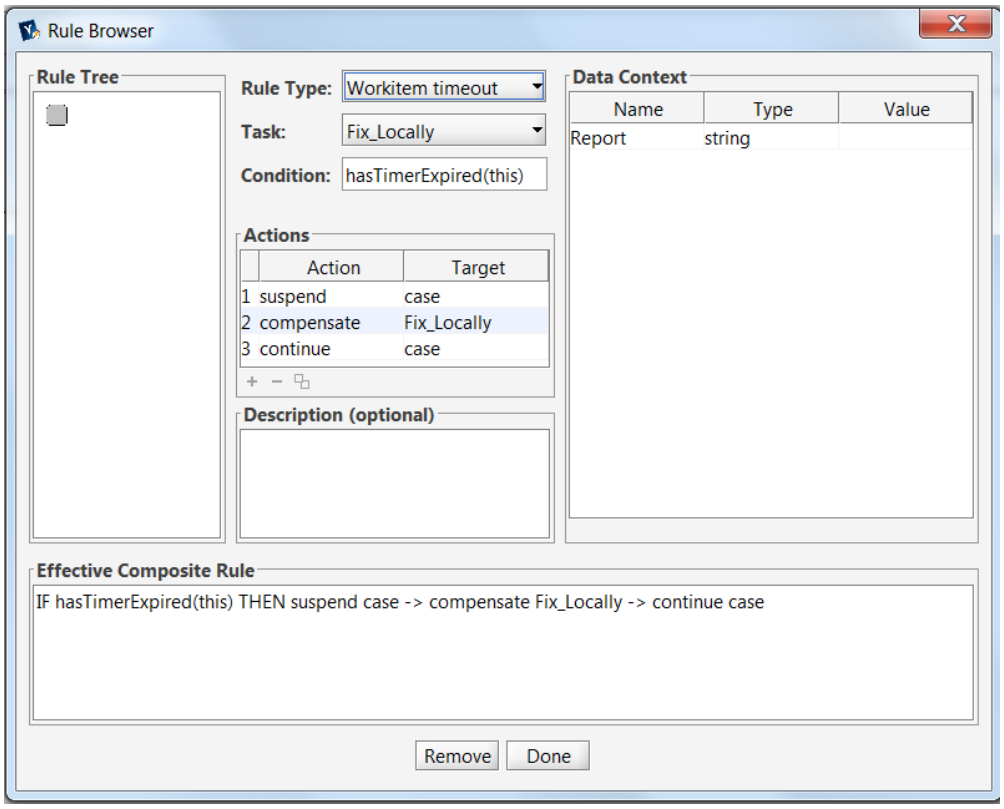

Figure 16: ‘Restart a task after a timeout’ rule definition.

4) “**Recall all terminals**”. In the *Fix_Major_Issue* process, the result of a vendor investigation can be a decision to notify all customers and recall all terminals. This is achieved by defining the rule depicted in Figure 17 (*Recall* worklet notifies customers about the recall).

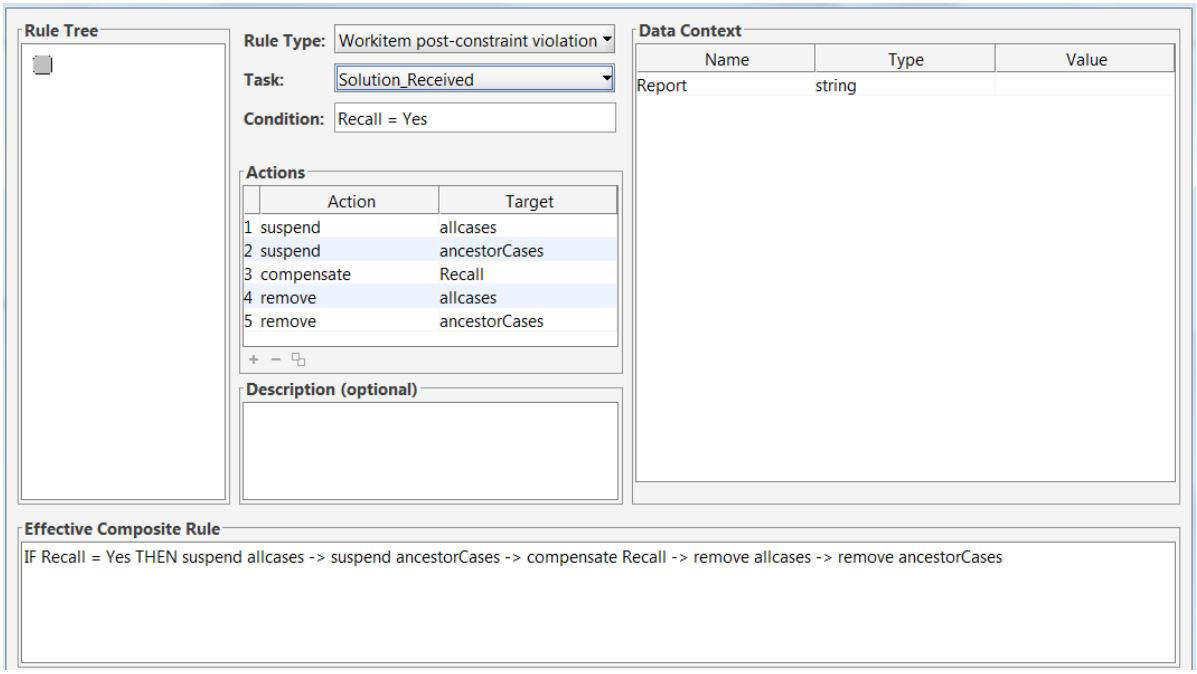

Figure 17: 'Recall all terminals' rule definition.

5) '**A minor issue is resolved during an early stage of the process'**. In the *Fix_Minor_Issue* process, if an engineer is able to fix an issue during the *Initial_Check* step, then the case is removed, as depicted in Figure 18.

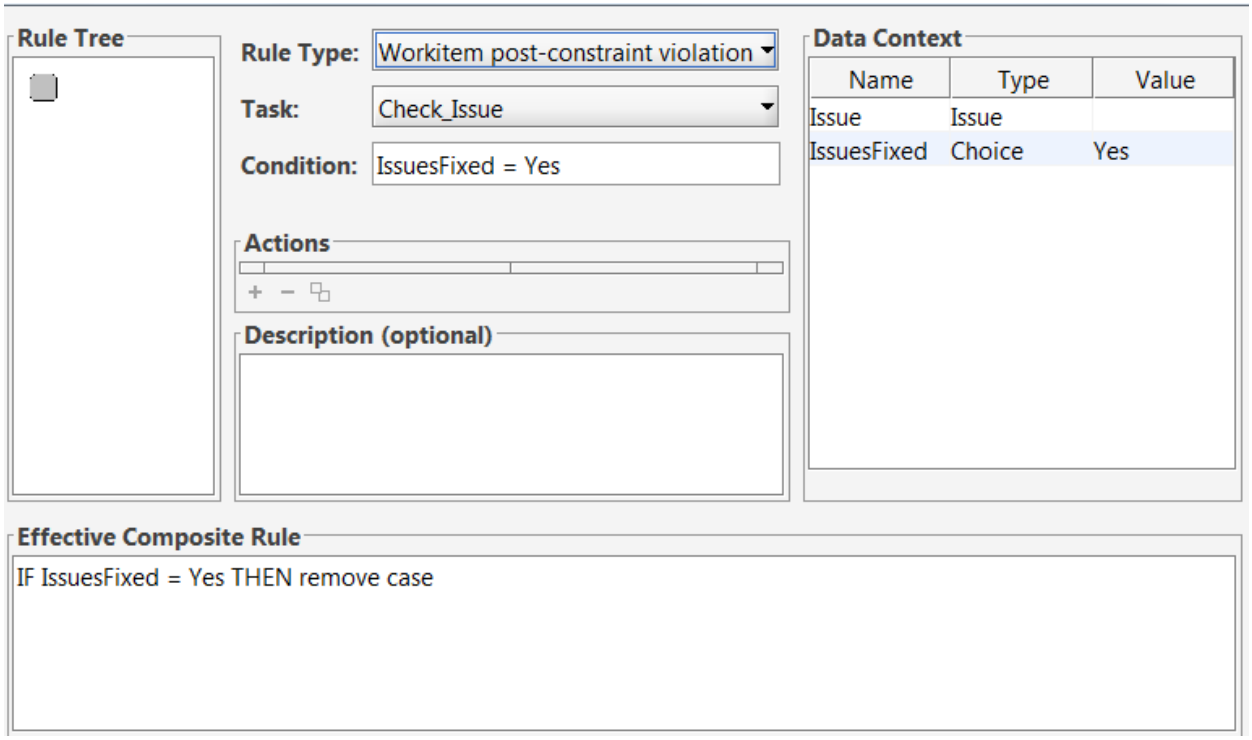

Figure 18: 'A minor issue is resolved at an early stage' rule definition.

6) '**An issue is reclassified as major**'. In the *Fix_Minor_Issue* process, an engineer can decide that the issue is a major issue. In this case *Fix_Minor_Issue* case has to be cancelled and *Fix_Major_Issue* process has to be started instead. The rule definition is depicted in Figure 19.

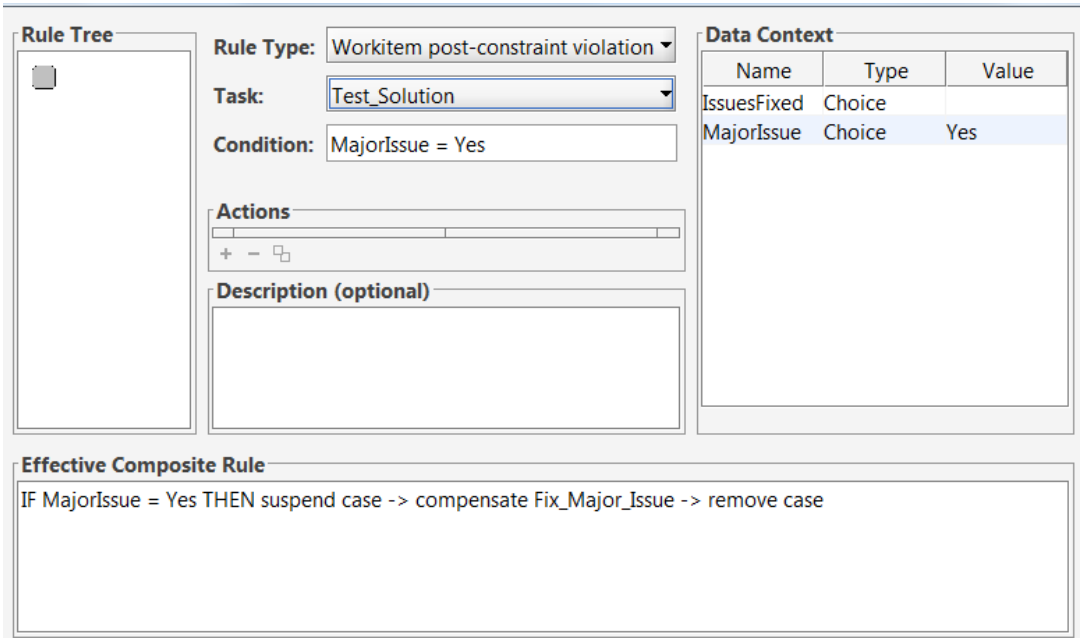

Figure 19: 'An issue is reclassified as major' rule definition.

Cost service

Cost model for the process (*AddTerminalCostModel.xml*) specifies wages for resources executing the process. Figure 20 depicts an example of cost definition associated with resource *SeniorEngineer1* executing task *Load_Keys*.

```
<driver>
        <metadata>
        <name>Senior Engineer</name>
        <description/>
        <type/>
    </metadata>
    <facets>
        <facet aspect="task">
        <name>Load_Keys</name>
        </facet>
        <facet aspect="resource">
        <name>SeniorEngineer1</name>
        </facet>
    </facets>
        <costtypes>
        <costtype>Wages Senior Engineer</costtype>
    </costtypes>
    <unitcost>
        <amount>70</amount>
        <currency>AUD</currency>
        <unit>hour</unit>
        <status>busy</status>
    </unitcost>
</driver>
```

Figure 20: Cost definition for resource SeniorEngineer1 executing task Load_Keys.

The cost model is used to allocate task *Load_Keys* to the cheapest resource as depicted in Figure 21.

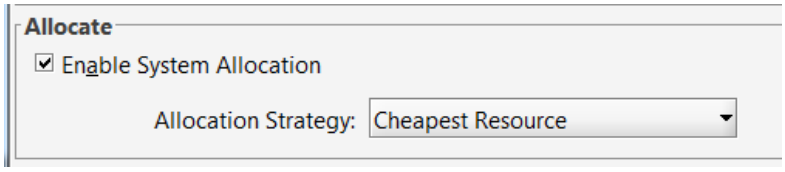

Figure 21: Resource allocation strategy for task Load_Keys.

The cost model is also used to calculate the overall cost of the case after task *Quality_Check* is completed. If the cost exceeds a threshold (50), task *Update_Budget* is executed (Figure 22).

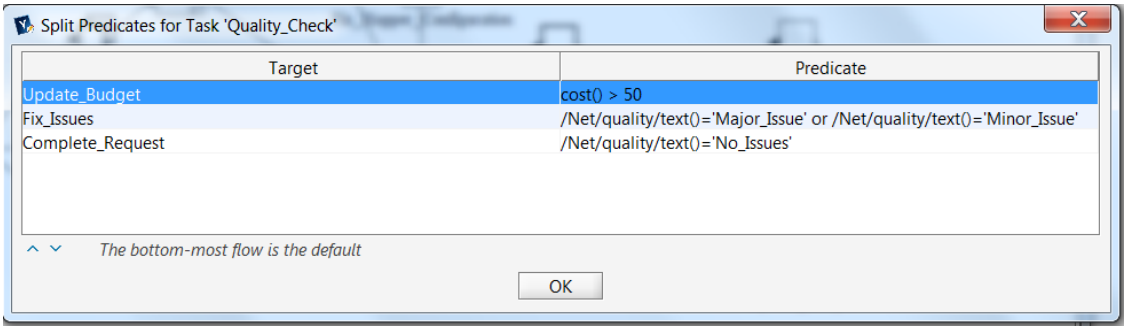

Figure 22: Split Predicates for task 'Quality_Check'.