



## A4.D3.1 – V1 of Dynamic Validation Prototype

K. Androutsopoulos, C.Ballas, C. Kloukinas, K. Mahbub, G. Spanoudakis

<b>Document Number</b>	A4.D3.1
<b>Document Title</b>	V1 of Dynamic Validation Prototype
<b>Version</b>	1.2
<b>Status</b>	Final
<b>Work Package</b>	WP 4.3
<b>Deliverable Type</b>	Prototype
<b>Contractual Date of Delivery</b>	31 December 2006
<b>Actual Date of Delivery</b>	8 February 2006
<b>Responsible Unit</b>	CUL
<b>Contributors</b>	CUL
<b>Keyword List</b>	S&D Monitoring Tool
<b>Dissemination level</b>	PU

## Change History

Version	Date	Status	Author (Unit)	Description
0.1	1/2/07	Draft	K. Androutsopoulos	Table of contents, indicative section contents
0.2	4/2/07	Draft	G. Spanoudakis	Added section 2.
0.3	5/2/07	Draft	C. Kloukinas	Added section 3.3/4/6/7, 4 + corrections
0.4	6/2/07	Draft	C. Ballas	Sections 3.3, 5
0.5	6/2/07	Draft	K. Mahbub	Sections 3.2, 3.4, 3.6
1.0	8/2/07	Final	C. Kloukinas	Sections 1, 4, 6
1.1	12/2/07	Final	C. Kloukinas	Changes to make document compliant with the quality plan
1.2	12/2/07	Final	K. Androutsopoulos	Changes to make document compliant with the quality plan

## **Executive Summary**

This document complements the code which has been delivered as the deliverable A4.D3.1. It contains a description of the first version of the Dynamic Validation Prototype which has been delivered, starting from the initial version of it and describing which architectural and other changes have been performed. It then goes through some of the limitations of the current version, some of which will be remedied in the second version of the Dynamic Validation Prototype (deliverable A4.D3.3).

Finally, the present document contains an installation and usage guide for the code which has been delivered in A4.D3.1.

## Table of Contents

1. Introduction .....	6
2. Original Monitoring Environment .....	7
2.1. Overview .....	7
2.2. Event Calculus .....	8
2.3. EC-Assertion .....	9
3. DVPv1 .....	11
3.1. Overview .....	11
3.2. New Form of EC Assertion Rules .....	12
3.2.1. The XML Schema for Monitoring Rules .....	12
3.2.2. Example of Monitoring Rules Expressed in XML .....	26
3.3. Communication with Event Collectors .....	30
3.3.1. New Event Structure & Event Buffer .....	30
3.3.2. Complex Object Types .....	31
3.3.3. User-Defined Complex Object Types .....	31
3.3.4. Support for Type Inheritance .....	31
3.3.5. Native Type Generator (NTG) .....	31
3.4. Fluent Modelling .....	39
3.5. Unification/Relations of objects .....	39
3.6. Support for Past-Time Formulae .....	39
3.6.1. Overview .....	39
3.6.2. Method for Checking Past-Time Predicates .....	42
3.7. Support for HoldsAt Predicates .....	43
3.7.1. Fluent Initiates and Terminates Database .....	43
3.7.2. Query Generation for HoldsAt .....	43
4. Limitations .....	44
4.1. Support for Future-Time HoldsAt Predicates .....	44
4.2. Lack of Multiple Inheritance .....	44
4.3. Support for Logical Clocks .....	44
4.4. Support for Non-Determinism .....	45
4.5. Support for a Full Object-Oriented Database .....	45
4.6. Minimisation of Past-Time Predicates in the DB .....	45
4.7. Calling External Functions .....	45

4.8. Supporting Internal Alarms .....	45
5. Installation and Usage Guide.....	46
5.1. Required Software .....	46
5.2. Installation.....	46
5.3. How to Use the DVP.....	46
5.3.1. The Data Analyzer.....	46
5.3.2. The Monitoring Manager .....	46
5.4. Example of Using the DVP Together With an Event Collector.....	53
6. Conclusion.....	54

# 1. Introduction

---

This report is part of the SERENITY A4.D3.1 deliverable and describes the implementation of the 1<sup>st</sup> version of the dynamic validation prototype of SERENITY. It also gives guidelines on how to install and use it in conjunction with the event captors that were developed and documented in deliverable A4.D2.2 [1]. It should be noted that, in addition to this report, A4.D3.1 includes:

- the source code of the 1<sup>st</sup> version of the dynamic validation prototype of SERENITY
- Example specifications of security and dependability properties that can be monitored using this prototype.
- Mock up implementations of systems that could be monitored using the formulae provided in (2).
- A set of event captors that need to be installed in order to generate the events required for monitoring the formulae described in (2).

The 1<sup>st</sup> version of the dynamic validation prototype, shortly referred to as *DVPv1* in the rest of this report has been based on an Event Calculus [4] monitoring engine that had been developed by City University and is described in [3][6][2]. DVPv1 extends this engine considerably by supporting the monitoring of past-time event calculus formulae, the use of variables in event calculus formulae which may be of arbitrary object types, the modelling of fluents which describe the state of a monitored system and its environment using object types, and reasoning about the truth value of such fluents on the basis of primitive fluent initiation and termination facts. These extensions were all necessary in order to support the monitoring of types of formulae which are necessary in order to specify basic security properties in Event Calculus and provide a more complete coverage of this temporal logic language.

The rest of this report is structured as follows. In Section 2, we describe the general architecture and functionality of the core monitoring engine upon which the development of DVPv1 has been based. In Section 3, we describe the extensions that we have introduced to this engine as part of the implementation of DVPv1. In Section 4, we provide an overview of the limitations of the current implementation of DVPv1 and make an initial assessment of the need to support them in the subsequent version of the dynamic validation prototype which is due in month 18. In Section 5, we provide guidelines for the installation and use of DVPv1. Finally, in Section 6 we provide some concluding remarks for DVPv1.

## 2. Original Monitoring Environment

### 2.1. Overview

The original environment that DVPv1 is based on was developed to monitor the operation of service based systems. This environment supports the monitoring of service based systems that are implemented by executable workflows which interact with web-services. More specifically, the monitoring environment supports workflows that are expressed in BPEL and assumes that there is some middleware platform that executes the workflow and can emit events regarding its execution. The properties that this environment can monitor are expressed in a language called *EC-Assertion* that is based on Event Calculus [4] and is specified by an XML schema. At runtime the environment checks the properties specified in EC-Assertion against the stream of events which are captured from the middleware platform that executes the service based system workflow and reports violations of these formulae. The monitoring environment is also capable of producing derived events and check whether the monitored properties are satisfied against the set of both the recorded and the derived events. The derivation of events is based on a special type of EC-Assertion formulae which are called *assumptions*.

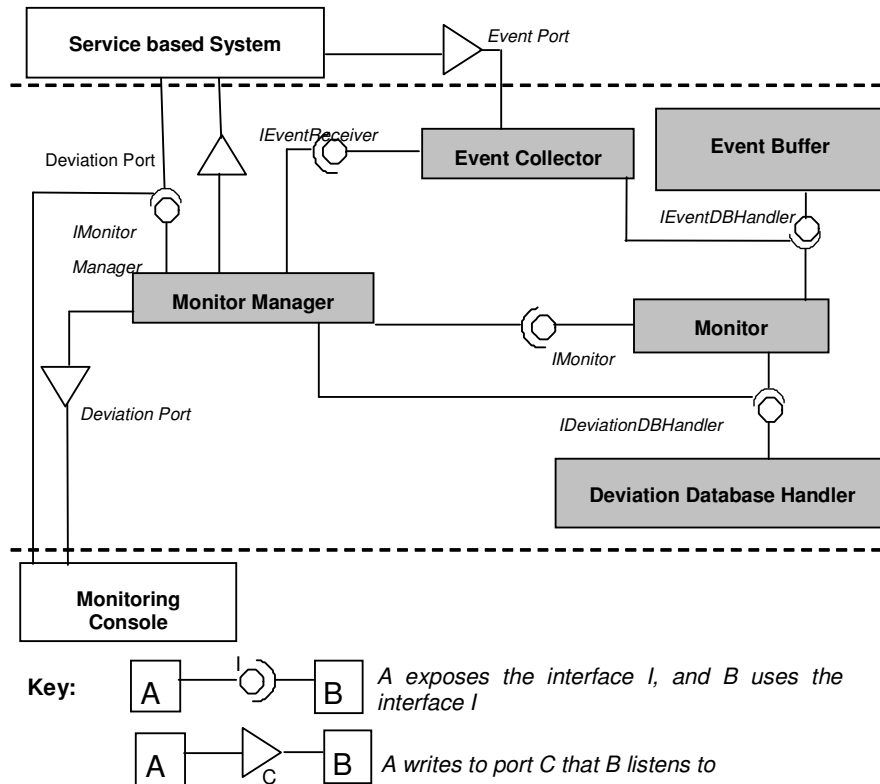


Figure 1 – Architecture of Original Monitoring Environment

Figure 1 shows the architecture of the original monitoring environment that DVPv1 is based on. As shown in the figure, this environment consists of a *monitoring manager*, an *event collector*, a *monitor*, an *event database*, a *deviation database* and a *monitoring console*.

The *monitoring manager* is the component that has responsibility for initiating, coordinating and reporting the results of the monitoring process. Once it receives a request for starting a monitoring activity, it checks whether it is possible to monitor the requested properties and, if it can, it starts an event collector to capture events from the SBS execution environment and passes to it the events that should be collected. It also sends to the monitor the formulae to be checked.

The *event collector* polls the event port of the SBS execution environment to get the stream of events sent to this port. After receiving an event, the event collector identifies its type and, if it is relevant to the properties which are being monitored, it sends it to the *event buffer*. All the events which are not relevant to the monitoring of the requested formulae or the assumptions which are used in order to derive information are ignored.

The *monitor* retrieves the events which are recorded in the event buffer during the operation of the SBS system in the order of their occurrence, derives other possible events that may have happened without being recorded (based on assumptions set for an SBS system), and checks if the recorded and derived events are compliant with the properties being monitored. In cases where the recorded and derived events are not consistent with properties being monitored, the monitor records the deviation in a *deviation database*. The monitoring manager polls the deviation database of the framework at regular time intervals to check if there have been any deviations detected with respect to the given set of properties and reports them to the monitoring console of the environment.

Finally, the environment incorporates a *monitoring console* that gives access to the monitoring service to human users. The console incorporates a *deviation viewer* that displays the deviations from the monitored properties. It also supports the selection of the properties to be monitored from sets of properties that have been predefined in EC-Assertion, enables the user to suspend and re-initiate monitoring, and gives access to the entire set of events that have been recorded during the operation of the SBS system.

## 2.2. Event Calculus

The event calculus (EC) is a first-order temporal formal language that can be used to specify properties of dynamic systems which change over time. Such properties are specified in terms of *events* and *fluents*.

An event in EC is something that occurs at a specific instance of time (e.g., invocation of an operation) and may change the state of a system. Fluents are conditions regarding the state of a system and are initiated and terminated by events. A fluent may, for example, signify that a specific system variable has a particular value at a specific instance of time.

The occurrence of an event is represented by the predicate  $\text{Happens}(e,t,\mathfrak{R}(t_1,t_2))$ . This predicate signifies that an instantaneous event  $e$  occurs at some time  $t$  within the time range  $\mathfrak{R}(t_1,t_2)$ . The boundaries of  $\mathfrak{R}(t_1,t_2)$  can be specified by using either time constants or arithmetic expressions over the time variables of other predicates in an EC formula.

The initiation of a fluent is signified by the EC predicate  $\text{Initiates}(e,f,t)$  whose meaning is that a fluent  $f$  starts to hold after the event  $e$  at time  $t$ . The termination of a fluent is signified by the EC predicate  $\text{Terminates}(e,f,t)$  whose meaning is that a fluent  $f$  ceases to hold after the event  $e$  occurs at



time  $t$ . An EC formula may also use the predicates  $\text{Initially}(f)$  and  $\text{HoldsAt}(f,t)$  to signify that a fluent  $f$  holds at the start of the operation of a system and that  $f$  holds at time  $t$ , respectively.

Event calculus defines a set of axioms that can be used to determine when a fluent holds based on initiation and termination events which may have occurred regarding this fluent. These axioms are listed in Table 1.

(EC1)	$\text{Clipped}(t1, f, t2) \Leftarrow (\exists e, t) \text{Happens}(e, t, \mathfrak{R}(t1, t2)) \wedge \text{Terminates}(e, f, t)$
(EC2)	$\text{HoldsAt}(f, t) \Leftarrow \text{Initially}(f) \wedge \neg \text{Clipped}(0, f, t)$
(EC3)	$\text{HoldsAt}(f, t) \Leftarrow (\exists e, t1) \text{Happens}(e, t, \mathfrak{R}(t1, t)) \wedge \text{Initiates}(e, f, t1) \wedge \neg \text{Clipped}(t1, f, t)$
(EC4)	$\text{Happens}(e, t, \mathfrak{R}(t1, t2)) \Rightarrow (t1 \leq t2) \wedge (t1 \leq t) \wedge (t \leq t2)$

**Table 1 – Axioms of Event Calculus**

The axiom *EC1* in Table 1 states that a fluent  $f$  is clipped (i.e., ceases to hold) within the time range from  $t1$  to  $t2$ , if an event  $e$  occurs at some time point  $t$  within this range and  $e$  terminates  $f$ . The axiom *EC2* states that a fluent  $f$  holds at time  $t$ , if it held at time 0 and has not been terminated between 0 and  $t$ . The axiom *EC3* states that a fluent  $f$  holds at time  $t$ , if an event  $e$  has occurred at some time point  $t1$  before  $t$  which initiated  $f$  at  $t1$  and  $f$  has not been clipped between  $t1$  and  $t$ . Finally, the axiom *EC4* states that the time range in a *Happens* predicate is inclusive of its boundaries.

### 2.3. EC-Assertion

The original form of EC-Assertion defined special types of fluents and events that were relevant to the operation of a service based system. More specifically, fluents were specified by expressions of the form:

$$\text{valueOf}(\text{fluent\_expression}, \text{value\_expression})$$

whose meaning was that the fluent identified by *fluent\_expression* had the value *value\_expression*. Furthermore, events in EC-Assertion represented exchanges of messages between the workflow of an SBS system and the web services interacting with it. Such messages could either invoke operations in services or the workflow or return results following the execution of an operation. Thus, depending on their sender and receiver, events in the original form of EC-Assertion could be:

- Service operation invocation events signifying the invocation of an operation in one of the partner services of an SBS system by its workflow;
- Service operation reply events signifying the return from the execution of an operation that has been invoked by the workflow of an SBS in one of its partner services;
- SBS operation invocation events signifying the invocation of an operation in the workflow of an SBS by one of its partner services; or
- SBS operation reply events signifying the reply following the execution of an operation that was invoked by a partner service in the composition process of an SBS.

In addition to using special types of events and fluents, a formula in the original form of EC-Assertion that is supported by the monitoring environment is valid only if:

- It specifies boundaries for the time ranges  $\mathfrak{R}(LB,UB)$  which appear in the Happens predicates
- For any variable  $t$  in Happens predicates that is existentially quantified, at least one of LB and UB must be specified. These boundaries can be specified by using: (i) constant time indicators or (ii) arithmetic expressions of time variables  $t'$  which appear in Happens predicates of the same formula provided that the latter variables are universally quantified, and that appears in their scope. If  $t$  is a universally quantified variable both LB and UB must be specified. Happens predicates with unrestricted universally quantified time variables take the form  $\text{Happens}(e,t,\mathfrak{R}(t,t))$ . These predicates express events whose time of occurrence is not constrained.
- The time variables of all the predicates in the formula which include existentially quantified non-time variables, take values in time ranges with fixed boundaries. These restrictions guarantee the ability to check the satisfiability of formulae.

Finally formulae in *EC-Assertion* can use the predicates  $<$  and  $=$  to express time conditions (the predicate  $t1 < t2$  is true if  $t1$  is a time instance that occurred before  $t2$ , and the predicate  $t1 = t2$  is true if  $t1$  is a time instance that is equal to  $t2$ ) and to compare values of different variables.

As an example of a property that could be specified in the original form of EC-Assertion consider the property specified by the following formula:

```

 $\forall$    _X: String; t1, t2:Time
      Happens(ic:S1:op(_X),t1,  $\mathfrak{R}(t1,t1)$ )  $\Rightarrow$  Happens(ir:S1:op(_X),t2,  $\mathfrak{R}(t1,t1+100_u)$ )

```

The above formula specifies that if an event  $ic:S1:op(_X)$  that signifies the call of the operation  $op(_X)$  in service S1 occurs at some time point  $t1$  then an event  $ic:S1:op(_X)$  signifying the response of S1 back to the caller of  $op(_X)$  should occur within 100 time units after  $t1$ .

## 3. DVPv1

---

### 3.1. Overview

DVPv1 has been developed by extending the monitoring environment we overviewed in Section 2 in order to provide support for:

- The communication with the event captors that we have developed in SERENITY for observing events from different types of systems and the structure that they use to represent and communicate these events (see deliverable A4.D2.2 [1])
- The use of object variables in EC-Assertion formulae.
- The use of arbitrary relations between objects of complex types (or parts of such objects) as fluents.
- The monitoring of past-time EC-Assertion formulae.
- Reasoning about the truth value of HoldsAt predicates in EC-Assertion formulae.

The above extensions were necessary in order to make the original monitoring environment capable of monitoring: (i) the operations of generic software systems as opposed to service based systems only that the original environment was able to monitor, and (ii) basic security and dependability properties such as confidentiality and integrity which as we show in [5] can be expressed by past-time EC-Assertion formulae.

In implementing DVPv1, we have also altered the architecture of the original monitoring framework illustrated in Figure 1. The new architecture is illustrated in Figure 2. The most significant alteration of this architecture is related to the communication of events from event captors to the framework. The event buffer becomes a subcomponent of the monitor manager because the event collector is not limited to receiving events from just web services (by making calls to web services). Instead, the event collector can receive events from more general types of systems via TCP/IP sockets as well as from web services. Alterations also occur with the internal components of the monitor, i.e. three additional subcomponents have been implemented. The *Native Type Generator (NTG)* is a component that receives the events in XML (as a string) and creates Java objects for the event and its elements (e.g. operation parameters, source, receiver, etc.) which can then be used by the monitor to perform the checking (via unification). There are also two in-memory databases that are used for checking past-time formulae and for supporting the evaluation of the HoldsAt predicate: *Database I* that keeps a record of the past-time events; and *Database II* that keeps a record of any Initiates and Terminates predicates.

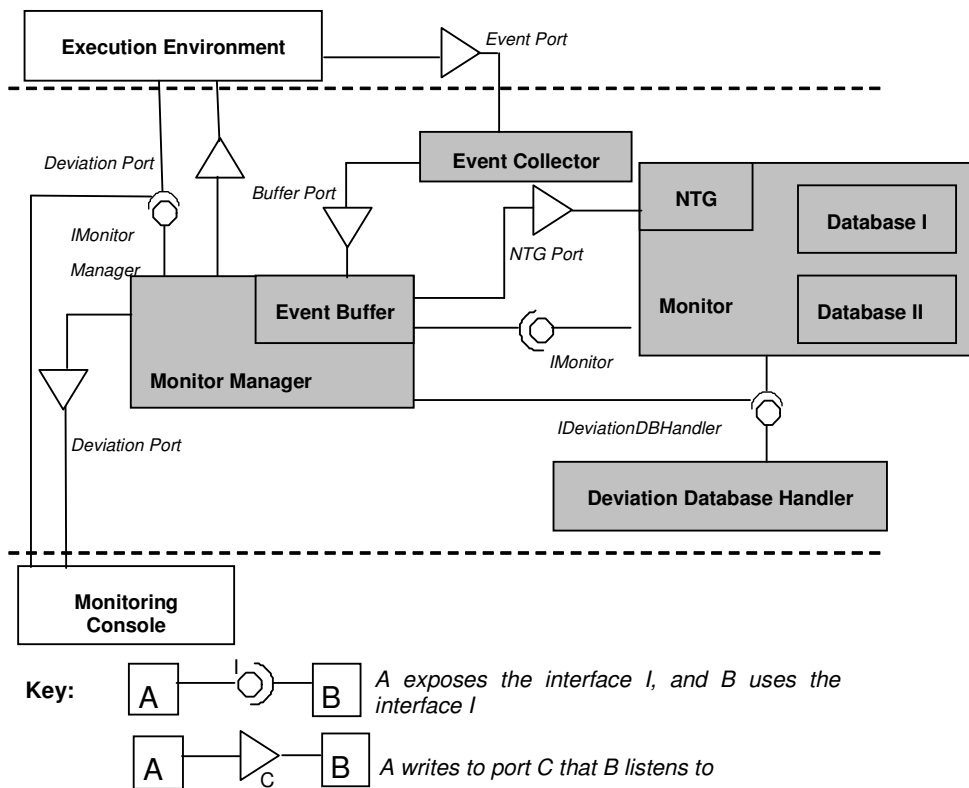


Figure 2 – Architecture of New Monitoring Environment

## 3.2. New Form of EC Assertion Rules

### 3.2.1. The XML Schema for Monitoring Rules

The structure of a monitoring rule is defined by the complex XML type called *formulaType*. It has two attributes: *formulaid* for identifying the formula, and *forChecking*, a Boolean used to distinguish between assumptions and rules. *formulaType* consists of the following child elements:

1. At least one *quantification* element, which is used to specify the quantification of variables in an EC formula. It is of type *quantificationType*, which is a complex type and consists of a *quantifier* element, to represent the quantifier (i.e. existential or universal), and a choice of variables that can be quantified, i.e. *regularVariable* (all other variables except for time variables) or *timeVariables*.
2. Zero or one *body* element, which specifies the expression on the RHS of the implication (if any), i.e. the body of the formula. It is of type *bodyHeadType*, a complex type that consists of the following sequence of child elements:

- a. A *predicate* element that is used to define the predicate in the formula and whose type is *predicateType*. *predicateType* is a complex type that has two attributes: *negated*, a Boolean used to indicate if a predicate is negated and whose default value is false, and *unconstrained*, a Boolean that is true if the predicate is unconstrained and whose default value is false. It also consists of the following child elements:
  - i. *happens*, which is of complex type *happensType*, that consists of the following sequence of elements:
    - *event* that is of type *eventType* for representing the event. This type is a complex type and consists of the following child elements: *eventID* of type *String* for identifying uniquely the event, *sender* of type *variableType* for specifying the agent that sends the message, *receiver* of *variableType* for specifying the agent that receives the message, *status* of type *String* for representing the processing status of an event, *oper* of type *operationType* for representing the operation signature that the event invokes or reports the result of and *source* of type *String* for specifying the agent that provided information about the event. The complex type *variableType* is explain later ( b,i, 8<sup>th</sup> bullet point). The complex type *operationType* consists of the following sequence of child elements: *opName* of type *String* for defining the name of the operation and zero or one *op\_args* of type *String* for defining the possible argument of an operation. See Figure 3.
    - *timeVar* is of complex type *timevariableType* that represents the time variable. The type *timevariableType* consists of the following child elements: *varName* for specifying the name of the variable, *varType* for specifying the type of the variable, and zero or one value element for specifying the value of the variable.
    - *fromTime* is of type *TimeExpression* and represents the starting time of the time range within which the formula should hold. *TimeExpression* consists of: a *time* element that is of type *timevariableType* that has been described above; and a choice of time operators, namely *plusTime* that is of type *timevariableType*, *minusTime* that is of type *timevariableType*, *plus* and *minus* which are both of *decimal* type.
    - *toTime* is of type *TimeExpression* and represent the finishing time of the time range within which the formula should hold. *TimeExpression* has been described in detail above.
  - ii. *initiates*, which is of complex type *initiatesType* that consists of the following child elements:
    - *event* that is of type *eventType* for representing the event, as described above.
    - *fluent* is of type *fluentType* and it distinguishes between the different types of fluents that can be described in the formula. *fluentType* is a complex type and it defines one of the following relations:

- A generic relation between  $n$  objects. In this case the complex type *fluentType* consists of a list of variables and the relation name is specified by the value of the attribute "name" of the fluent element.
- A predefined relation where a variable that is given at the *target* (i.e. the first argument) is updated with the value or either a variable at the *source* (i.e. the second argument) or with the return value of an operation that is called. The complex type *fluentType*, therefore consists of: a *target* and a *source* element. The types of these elements consequently consist of a *variable* element, and in the case of the source, or an *operationCall* element. In this case the value of the attribute " name" of the fluent element must be "valueOf".
  - *timeVar* is of complex type *timevariableType* that represents the time variable.
- ii. *holdsAt* is of type *holdsatType* that consists of the following sequence of elements:
  - *fluent* that is of type *fluentType* (as described for the initiated predicate).
  - *timeVar* that is of type *timevariableType*(as described for the initiated predicate).
- iii. *initially* is of type *holdsatType*, which is described above.
- iv. *terminates* is of type *terminatesType* that is a complex type that consists of the following child elements:
  - *event* is of type *eventType* that has been previously described.
  - *fluent* is of type *fluentType* that has been previously described.
  - *timeVar* is of type *timevariableType* that has been previously described.
- b. *relationalPredicate* is of complex type *relationPredicateType* that specifies the possible relations between two variables in the formula. This type has the following child elements:
  - v. a choice of the following elements:
    - *equalTo*
    - *notEqualTo*
    - *lessThan*
    - *greaterThan*

- `lessThanOrEqualTo`
- `greaterThanOrEqualTo`

which are all of complex type *varRelationType* that consists of two elements: *operand1* and *operand2* of type *operandType*. The complex type *operandType* consists of the following choice of elements (only one of these elements will be represented):

- *operationCall* that is of type *operationCallType* that has a sequence of child elements: *name* of type *String*, zero or one partner of type *String* and zero or more (unbounded) variable elements of type *variableType*, which is described below.
- *variable* that is of type *variableType*. This type is a complex type that has two attributes: *persistent* that indicates whether the value of the variable is the same throughout all instances (like static variables in Java) and *forMatching* that distinguishes between internal and external variables (i.e. its value is false for internal variables). Also, the type consists of the following child elements: *varName* that is of type *String*, and either a *varType* and *value* element, both of type *String*, or an *array* element of type *arrayType* with elements that describe the array structure. And,
- *constant* that is of type *constantType* for describing constants. This type consists of two elements: *name* and *value* elements which are both of type *String*.
- *timeVar* is of type *timevariableType* that has been previously described.

c. a possible sequence of an *operator* and a choice of either:

- i. a *predicate* that is of type *predicateType* that has been explained earlier,
- ii. a *timePredicate* that is of type *timepredicateType*. This element is used to express a relation between two time variables in the formula. It has a choice of the following child elements: *timeEqualTo*, *timeNotEqualTo*, *timeLessThan*, *timeGreaterThan*, *timeLessThanOrEqualTo*, *timeGreaterThanOrEqualTo*, all of complex type *TimeRelation* that consist of two elements: *timeVar1* and *timeVar2* of type *TimeExpression* that has been described earlier. Or
- iii. a *relationPredicate* that is of type *relationPredicateType* that has been explained earlier.

A *head* element which is of type *bodyHeadType*, which is described above.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

```

<xs:element name="formulasdesc" type="formulasdescType"/>

<xs:complexType name="formulasdescType">
  <xs:sequence>
    <xs:element name="datatypes" type="xs:anyURI" minOccurs="0"
maxOccurs="unbounded"/>
    <xs:element name="formulas" type="formulasType"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="formulasType">
  <xs:sequence>
    <xs:element name="formula" type="formulaType" minOccurs="1"
maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="formulaType">
  <xs:sequence>
    <xs:element name="quantification" type="quantificationType" minOccurs="1"
maxOccurs="unbounded"/>
    <xs:element name="body" type="bodyHeadType" minOccurs="0"/>
    <xs:element name="head" type="bodyHeadType"/>
  </xs:sequence>
  <xs:attribute name="formulaId" type="xs:string" use="required"/>
  <xs:attribute name="forChecking" type="xs:boolean" default="true"/>
</xs:complexType>

<xs:complexType name="bodyHeadType">
  <xs:sequence>
    <xs:choice>
      <xs:element name="predicate" type="predicateType"/>
      <xs:element name="relationalPredicate" type="relationalPredicateType"/>
    </xs:choice>
  </xs:sequence>
</xs:complexType>

```



```
<xs:sequence minOccurs="0" maxOccurs="unbounded">
  <xs:element name="operator" type="logicalOperatorType"/>
  <xs:choice>
    <xs:element name="predicate" type="predicateType"/>
    <xs:element name="timePredicate" type="timePredicateType"/>
    <xs:element name="relationalPredicate" type="relationalPredicateType"/>
  </xs:choice>
</xs:sequence>
</xs:sequence>
</xs:complexType>

<xs:complexType name="predicateType">
  <xs:choice>
    <xs:element name="happens" type="happensType"/>
    <xs:element name="initiates" type="initiatesType"/>
    <xs:element name="holdsAt" type="holdsAtType"/>
    <xs:element name="initially" type="holdsAtType"/>
    <xs:element name="terminates" type="terminatesType"/>
    <xs:element name="clipped" type="clippedType"/>
    <xs:element name="declipped" type="declippedType"/>
  </xs:choice>
  <xs:attribute name="negated" type="xs:boolean" default="false"/>
  <xs:attribute name="unconstrained" type="xs:boolean" default="false"/>
</xs:complexType>

<xs:complexType name="timePredicateType">
  <xs:choice>
    <xs:element name="timeEqualTo" type="TimeRelation"/>
    <xs:element name="timeNotEqualTo" type="TimeRelation"/>
    <xs:element name="timeLessThan" type="TimeRelation"/>
    <xs:element name="timeGreaterThan" type="TimeRelation"/>
    <xs:element name="timeLessThanEqualTo" type="TimeRelation"/>
    <xs:element name="timeGreaterThanEqualTo" type="TimeRelation"/>
  </xs:choice>
</xs:complexType>
```

```
</xs:choice>
</xs:complexType>

<xs:complexType name="holdsAtType">
  <xs:sequence>
    <xs:element name="fluent" type="fluentType"/>
    <xs:element name="timeVar" type="timeVariableType"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="initiatesType">
  <xs:sequence>
    <xs:choice>
      <xs:element name="ir_term" type="irTermType"/>
      <xs:element name="rc_term" type="rcTermType"/>
      <xs:element name="as_term" type="asTermType"/>
    </xs:choice>
    <xs:element name="fluent" type="fluentType"/>
    <xs:element name="timeVar" type="timeVariableType"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="happensType">
  <xs:sequence>
    <xs:choice>
      <xs:element name="ic_term" type="icTermType"/>
      <xs:element name="ir_term" type="irTermType"/>
      <xs:element name="rc_term" type="rcTermType"/>
      <xs:element name="re_term" type="reTermType"/>
      <xs:element name="as_term" type="asTermType"/>
    </xs:choice>
    <xs:element name="timeVar" type="timeVariableType"/>
    <xs:element name="fromTime" type="TimeExpression"/>
  </xs:sequence>
</xs:complexType>
```

```
<xs:element name="toTime" type="TimeExpression"/>
</xs:sequence>
</xs:complexType>

<xs:complexType name="clippedType">
  <xs:sequence>
    <xs:element name="timeVar1" type="timeVariableType"/>
    <xs:element name="fluent" type="fluentType"/>
    <xs:element name="timeVar2" type="timeVariableType"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="declippedType">
  <xs:sequence>
    <xs:element name="timeVar1" type="timeVariableType"/>
    <xs:element name="fluent" type="fluentType"/>
    <xs:element name="timeVar2" type="timeVariableType"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="terminatesType">
  <xs:sequence>
    <xs:choice>
      <xs:element name="ir_term" type="irTermType"/>
      <xs:element name="rc_term" type="rcTermType"/>
      <xs:element name="as_term" type="asTermType"/>
    </xs:choice>
    <xs:element name="fluent" type="fluentType"/>
    <xs:element name="timeVar" type="timeVariableType"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="fluentType">
```

```
<xs:choice>
  <xs:sequence>
    <xs:element name="target">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="variable" type="variableType"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="source">
      <xs:complexType>
        <xs:choice>
          <xs:element name="variable" type="variableType"/>
          <xs:element name="operationCall" type="operationCallType"/>
        </xs:choice>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
  <xs:sequence>
    <xs:element name="variable" type="variableType" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:choice>
<xs:attribute name="name" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="quantificationType">
  <xs:sequence>
    <xs:element name="quantifier">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:pattern value="forall|existential"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

```
</xs:element>
<xs:choice >
  <xs:element name="regularVariable" type="variableType"/>
  <xs:element name="timeVariable" type="timeVariableType"/>
</xs:choice>
</xs:sequence>
</xs:complexType>

<xs:complexType name="icTermType">
  <xs:sequence>
    <xs:element name="operationName" type="xs:string"/>
    <xs:element name="partnerName" type="xs:string"/>
    <xs:element name="id" type="xs:string"/>
    <xs:element name="variable" type="variableType" minOccurs="0"
maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="irTermType">
  <xs:sequence>
    <xs:element name="operationName" type="xs:string"/>
    <xs:element name="partnerName" type="xs:string"/>
    <xs:element name="id" type="xs:string"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="rcTermType">
  <xs:sequence>
    <xs:element name="operationName" type="xs:string"/>
    <xs:element name="partnerName" type="xs:string"/>
    <xs:element name="id" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

```

<xs:complexType name="reTermType">
  <xs:sequence>
    <xs:element name="operationName" type="xs:string"/>
    <xs:element name="partnerName" type="xs:string"/>
    <xs:element name="id" type="xs:string"/>
    <xs:element name="variable" type="variableType" minOccurs="0"
maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="asTermType">
  <xs:sequence>
    <xs:element name="operationName" type="xs:string"/>
    <xs:element name="id" type="xs:string"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="variableType">
  <xs:sequence>
    <xs:element name="varName" type="xs:string"/>
    <xs:choice>
      <xs:sequence>
        <xs:element name="varType" type="xs:string"/>
        <xs:element name="value" type="xs:string" minOccurs="0"/>
      </xs:sequence>
      <xs:element name="array" type="arrayType"/>
    </xs:choice>
  </xs:sequence>
  <xs:attribute name="persistent" type="xs:boolean" default="false"/>
  <xs:attribute name="forMatching" type="xs:boolean" default="true"/>
</xs:complexType>

<xs:complexType name="expresionType">
  <xs:sequence>

```

```

<xs:element name="varName" type="xs:string"/>
<xs:choice>
  <xs:sequence>
    <xs:element name="varType" type="xs:string"/>
    <xs:choice>
      <xs:element name="value" type="xs:string"/>
      <xs:element name="fields" type="fieldType"/>
    </xs:choice>
  </xs:sequence>
  <xs:element name="array" type="arrayType"/>
</xs:choice>
</xs:sequence>
<xs:attribute name="persistent" type="xs:boolean" default="false"/>
<xs:attribute name="forMatching" type="xs:boolean" default="true"/>
</xs:complexType>

<xs:complexType name="fieldType">
  <xs:sequence>
    <xs:element name="field" type="xs:string" minOccurs="1" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="timeVariableType">
  <xs:sequence>
    <xs:element name="varName" type="xs:string"/>
    <xs:element name="varType" type="xs:string" fixed="TimeVariable"/>
    <xs:element name="value" type="xs:string" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>

<xs:simpleType name="logicalOperatorType">
  <xs:restriction base="xs:string">

```

```
<xs:pattern value="andlor"/>
</xs:restriction>
</xs:simpleType>

<xs:complexType name="TimeExpression">
  <xs:sequence>
    <xs:element name="time" type="timeVariableType"/>
    <xs:sequence minOccurs="0" maxOccurs="unbounded">
      <xs:choice>
        <xs:element name="plusTime" type="timeVariableType"/>
        <xs:element name="minusTime" type="timeVariableType"/>
        <xs:element name="plus" type="xs:decimal"/>
        <xs:element name="minus" type="xs:decimal"/>
      </xs:choice>
    </xs:sequence>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="TimeRelation">
  <xs:sequence>
    <xs:element name="timeVar1" type="TimeExpression"/>
    <xs:element name="timeVar2" type="TimeExpression"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="varRelationType">
  <xs:sequence>
    <xs:element name="operand1" type="operandType"/>
    <xs:element name="operand2" type="operandType"/>
  </xs:sequence>
</xs:complexType>
```



```

<xs:complexType name="relationalPredicateType">
  <xs:sequence>
    <xs:choice>
      <xs:element name="equal" type="varRelationType"/>
      <xs:element name="notEqualTo" type="varRelationType"/>
      <xs:element name="lessThan" type="varRelationType"/>
      <xs:element name="greaterThan" type="varRelationType"/>
      <xs:element name="lessThanEqualTo" type="varRelationType"/>
      <xs:element name="greaterThanEqualTo" type="varRelationType"/>
    </xs:choice>
    <xs:element name="timeVar" type="timeVariableType"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="operandType">
  <xs:choice>
    <xs:element name="operationCall" type="operationCallType"/>
    <xs:element name="expresion" type="expresionType"/>
    <xs:element name="constant" type="constantType"/>
  </xs:choice>
</xs:complexType>

<xs:complexType name="operationCallType">
  <xs:sequence>
    <xs:element name="name" type="xs:string"/>
    <xs:element name="partner" type="xs:string" minOccurs="0"/>
    <xs:element name="expresion" type="expresionType" minOccurs="0"
maxOccurs="unbounded"/>
    <xs:element name="operationCall" type="operationCallType" minOccurs="0"
maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="constantType">

```

```

<xs:sequence>
  <xs:element name="name" type="xs:string"/>
  <xs:element name="value" type="xs:string"/>
</xs:sequence>
</xs:complexType>

<xs:complexType name="arrayType">
  <xs:sequence>
    <xs:element name="type" type="xs:string"/>
    <xs:element name="value" type="arrayValueType" minOccurs="0"
maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="arrayValueType">
  <xs:sequence>
    <xs:element name="indexValue" type="xs:string"/>
    <xs:element name="cellValue" type="xs:string"/>
  </xs:sequence>
</xs:complexType>

</xs:schema>

```

**Figure 3 – XML Schema for EC-Assertion**

### 3.2.2. *Example of Monitoring Rules Expressed in XML*

In this section, we consider the pattern for a Mechanism for Optimistic Fair Exchange with Trusted Third Party, which is described in section 5.4 of this deliverable, and give an example of a rule that can be monitored for this pattern. The monitoring rule is derived from the requirement that TTP must be available. It should consider two cases, i.e. the case when Alice tries to communicate with TTP and the case when Bob tries to communicate with TTP. Therefore, two rules are required. We illustrate the second case, more specifically: if Bob sends a “solve” message to TTP, then TTP should respond with “send\_item” message within some time limit ( $t_1+t_u$  where  $t_1$  is the time when Bob sent the “solve” message). In event calculus we express this as follows:

$$\begin{aligned} &\forall \quad \_eID1, \_eID2, Bob\_ID, \_TTP\_ID: String; t1, t2:Time \\ &\quad \mathbf{Happens} (e(\_eID1, Bob\_ID, TTP\_ID, REQ-B, solve((Item\_A) Kal, Item\_B)), Bob\_ID), t1, \\ &\quad \mathfrak{R}(t1, t1) \Rightarrow \end{aligned}$$

**Happens** (e (\_eID2, TTP\_ID, Bob\_ID, RES-A, send\_item( (Item\_A) Ka1) Ka2) , t2,  $\mathfrak{R}(t1, t1+t_u)$ )

The XML document that describes the above monitoring rule is given in Table 2. Firstly, the quantification of the variables in the formula is represented in lines 7-32. Two types of variables are quantified, namely regular variables (any variable except for time variables) and time variables. Next, the body of the formula is represented in lines 33-78, i.e. the expression on the RHS of the implication. The body consists of the **Happens** predicate and its arguments, i.e. an event, a time variable and a time range. The event is represented in lines 36-55. The time variable has been specified in lines 56-59 and the time range in lines 60-75. Finally, the head of the formula (i.e. the expression on the LHS of the implication) is represented in lines 79-120. This also consists of a **Happens** predicate with an event, a time variable and a time range, and thus is represented similarly to the body of the formula.

```

01 <?xml version="1.0" encoding="UTF-8"?>
02 <formulae xmlns="http://tempuri.org/ec/formula"
03     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
04     xsi:schemaLocation="http://tempuri.org/ec/formula
05 file:/Z:/Serenity/A5%20contribution%20-%20September06/EC-Assertion6.xsd"
06     formulaId="">
07     <quantification>
08         <quantifier>universal</quantifier>
09         <regularVariable>
10             <varName>Bob_ID</varName>
11             <varType>String</varType>
12         </regularVariable>
13         <regularVariable>
14             <varName>TTP_ID</varName>
15             <varType>String</varType>
16         </regularVariable>
17         <regularVariable>
18             <varName>_eID1</varName>
19             <varType>String</varType>
20         </regularVariable>
21         <regularVariable>
22             <varName>_eID2</varName>
23             <varType>String</varType>
24         </regularVariable>
25         <timeVariable>
26             <varName>t1</varName>
27             <varType>Time</varType>
28         </timeVariable>
29     </quantification>
30     <body>
31         <event>
32             <name>send_item</name>
33             <arguments>
34                 <argument>
35                     <name>Item_A</name>
36                     <value>Ka1</value>
37                 </argument>
38                 <argument>
39                     <name>Ka2</name>
40                     <value>Ka2</value>
41                 </argument>
42             </arguments>
43         </event>
44         <timeVariable>
45             <name>t2</name>
46             <value>t2</value>
47         </timeVariable>
48         <timeRange>
49             <start>t1</start>
50             <end>t1+t_u</end>
51         </timeRange>
52     </body>
53 </formulae>

```

28	<code>&lt;varName&gt;t2&lt;/varName&gt;</code>
29	<code>&lt;varType&gt;Time&lt;/varType&gt;</code>
30	<code>&lt;/timeVariable&gt;</code>
31	<code>&lt;/quantification&gt;</code>
32	<code>&lt;body&gt;</code>
33	<code>&lt;predicate&gt;</code>
34	<code>&lt;happens&gt;</code>
35	<code>&lt;event&gt;</code>
36	<code>&lt;eventID&gt;_eID2&lt;/eventID&gt;</code>
37	<code>&lt;sender&gt;</code>
38	<code>&lt;varName&gt;TTP_ID&lt;/varName&gt;</code>
39	<code>&lt;varType&gt;String&lt;/varType&gt;</code>
40	<code>&lt;/sender&gt;</code>
41	<code>&lt;receiver&gt;</code>
42	<code>&lt;varName&gt;Bob_ID&lt;/varName&gt;</code>
43	<code>&lt;varType&gt;String&lt;/varType&gt;</code>
44	<code>&lt;/receiver&gt;</code>
45	<code>&lt;status&gt;RES-A&lt;/status&gt;</code>
46	<code>&lt;oper&gt;</code>
47	<code>&lt;opName&gt;send_item&lt;/opName&gt;</code>
48	<code>&lt;op_args&gt;</code>
49	<code>&lt;varName&gt;((Item_A)Ka1)Ka2&lt;/varName&gt;</code>
50	<code>&lt;varType&gt;String&lt;/varType&gt;</code>
51	<code>&lt;/op_args&gt;</code>
52	<code>&lt;/oper&gt;</code>
53	<code>&lt;source&gt;TTP_ID&lt;/source&gt;</code>
54	<code>&lt;/event&gt;</code>
55	<code>&lt;timeVar&gt;</code>
56	<code>&lt;varName&gt;t2&lt;/varName&gt;</code>
57	<code>&lt;varType&gt;Time&lt;/varType&gt;</code>
58	<code>&lt;/timeVar&gt;</code>
59	<code>&lt;fromTime&gt;</code>
60	<code>&lt;time&gt;</code>
61	<code>&lt;varName&gt;t1&lt;/varName&gt;</code>
62	<code>&lt;varType&gt;Time&lt;/varType&gt;</code>
63	<code>&lt;/time&gt;</code>
64	<code>&lt;/fromTime&gt;</code>
65	<code>&lt;toTime&gt;</code>

66	<code>&lt;time&gt;</code>
67	<code>    &lt;varName&gt;t1&lt;/varName&gt;</code>
68	<code>    &lt;varType&gt;Time&lt;/varType&gt;</code>
69	<code>&lt;/time&gt;</code>
70	<code>&lt;plusTime&gt;</code>
71	<code>    &lt;varName&gt;tu&lt;/varName&gt;</code>
72	<code>    &lt;varType&gt;Time&lt;/varType&gt;</code>
73	<code>&lt;/plusTime&gt;</code>
74	<code>&lt;/toTime&gt;</code>
75	<code>&lt;/happens&gt;</code>
76	<code>&lt;/predicate&gt;</code>
77	<code>&lt;/body&gt;</code>
78	<code>&lt;head&gt;</code>
79	<code>  &lt;predicate&gt;</code>
80	<happens>
81	<event>
82	<eventID>_eID1</eventID>
83	<sender>
84	<varName>Bob_ID</varName>
85	<varType>String</varType>
86	</sender>
87	<receiver>
88	<varName>TTP_ID</varName>
89	<varType>String</varType>
90	</receiver>
91	<status>REQ-B</status>
92	<oper>
93	<opName>Solve</opName>
94	<op_args>
95	<varName>((Item_A)Ka1, Item_B)</varName>
96	<varType>String</varType>
97	</op_args>
98	</oper>
99	<source>Bob_ID</source>
100	</event>
101	<timeVar>
102	<varName>t1</varName>
103	<varType>Time</varType>

104	<code>&lt;/timeVar&gt;</code>
105	<code>&lt;fromTime&gt;</code>
106	<code>    &lt;time&gt;</code>
107	<code>        &lt;varName&gt;t1&lt;/varName&gt;</code>
108	<code>        &lt;varType&gt;Time&lt;/varType&gt;</code>
109	<code>    &lt;/time&gt;</code>
110	<code>&lt;/fromTime&gt;</code>
111	<code>&lt;toTime&gt;</code>
112	<code>    &lt;time&gt;</code>
113	<code>        &lt;varName&gt;t1&lt;/varName&gt;</code>
114	<code>        &lt;varType&gt;Time&lt;/varType&gt;</code>
115	<code>    &lt;/time&gt;</code>
116	<code>&lt;/toTime&gt;</code>
117	<code>&lt;/happens&gt;</code>
118	<code>&lt;/predicate&gt;</code>
119	<code>&lt;/head&gt;</code>
120	<code>&lt;/formulae&gt;</code>
121	

Table 2 – XML document representing the rule that checks the availability of TTP

### 3.3. Communication with Event Collectors

#### 3.3.1. New Event Structure & Event Buffer

Since DVPv1 needs to events which are produced by different types of systems and not only from web-service based systems as its precursor, we have thoroughly changed the XML schema of the event structure so as to be able to represent event characteristics which were not present previously, e.g., process id, group id, mobile identification number, etc. We have also redesigned the event structure so that it can now hold more information. For example, events representing operation calls now carry information about the sender and the receiver of the call but also about the source component where this event has been collected from. We are also now capable of distinguishing between a request for an operation which has been received but not yet serviced (REQ-B – B for before) versus a request which has been serviced (REQ-A – A for after), while previously we were not able to distinguish between these two cases. Similarly for responses, we now have two characterisations – RES-B & RES-A, for responses which will be sent to the initiator of the operation and for responses which have already been sent to it.

Since the events are produced inside different types of application systems, DVPv1 can no longer assume that the event collectors will be communicating with the monitoring infrastructure using web-service calls. Therefore, we have changed the architecture of the system, as shown in Figure 2, so that the event collectors can use the much more basic communication framework of TCP/IP sockets to send the events as strings to the Event Buffer and ask the Monitor Manager to perform the web-service call itself.

### 3.3.2. *Complex Object Types*

Parts of the new event structure consist now of complex object types (e.g., the sender field of an event is now of type `entity`). The code has been extended so that these complex object types can be treated like any other EC formula variable type, allowing the user to specify new variables of these types and also ensuring that relations on these object types (e.g., comparison for equality) will work seamlessly, making sure that in each case we also consider the relations between the types of the objects represented by the variables.

In order for this change to be possible we changed a considerable amount of the internal code of the monitor engine. Up to its previous incarnation, all variables would be represented as a pair of type and value, where both fields were Java strings. Then, relations over these variables would use their string representation, thus causing a considerable delay in the evaluation of formulae.

In DVPv1 the code has been changed throughout, so that all variables are represented as normal Java objects and operations upon them do not need to translate from/to strings anymore, thus considerably speeding up the execution of the engine.

Along with the existence of these new types, we have also extended the tool to support access to the member fields of the objects belonging to these types, so that one can write expressions of the form `sender.IPAddress == "www.google.com"`, etc. The expressions for accessing object fields are allowed in all places where a normal variable would be allowed – parameters of operations, left and right-hand side of relations, etc.

### 3.3.3. *User-Defined Complex Object Types*

Along with the predefined complex object types which we have introduced, we are now allowing users to specify their own application-domain specific object types, to be used with their formulae. In order to do so, users must precede their formulae with the definitions of their types in the normal XML schema. These types then become available for use in the formulae, in exactly the same way that primitive types (e.g., `integer`, `string`) and predefined complex types (e.g., `entity` which is the type of the sender of an operation) are been used. So users can now specify new variables using their own types, specify relations over them, etc.

### 3.3.4. *Support for Type Inheritance*

The types that users are defining can also be sub-types of other types (either user-defined or predefined). So if the underlying system and S&D Pattern has used an object-oriented design, the users are now capable of representing this in their formulae. As a consequence, if an operation in a formula has been declared to receive a parameter of type `fubar`, then the formula will be activated even if the operation has been called with an argument whose type is a sub-type of `fubar`.

### 3.3.5. *Native Type Generator (NTG)*

The component in DVPv1 which is responsible for identifying user-defined complex object types is the Native Type Generator, marked as NTG in Figure 2. This component basically reads the XML file containing the EC formulae and parses the user-specified complex object types using the JAXB library. By doing so, it produces native Java classes which represent these new types, loads these classes at run-time and makes them available to the monitor for use as variable types.

As an example, let us consider the XML types `personinfo` and `subscriber` (which inherits from `personinfo`) as these are defined with the XSD schema presented in Table 3 and graphically shown in Figure 4. Using the JAXB compiler, we will obtain two Java classes representing these new

XML types, Personinfo.java and Subscriber.java, where class Subscriber will be defined as a subclass of class Personinfo – see Table 4 and Table 5.

```

<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc"
  jaxb:extensionBindingPrefixes="xjc"
  xmlns:jaxb="http://java.sun.com/xml/ns/jaxb" jaxb:version="1.0">

  <!--
    This uses the additional vendor specific features
    provided by the JAXB RI
    They are defined in the "http://java.sun.com/xml/ns/jaxb/xjc"
  namespace -->

  <xs:annotation>
    <xs:appinfo>
      <jaxb:globalBindings>

        <xjc:serializable uid="12343"/>

      </jaxb:globalBindings>
    </xs:appinfo>
  </xs:annotation>

  <!-- <xs:element name="employee" type="subscriber"/> -->

  <xs:complexType name="personinfo">
    <xs:sequence>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="subscriber">
    <xs:complexContent>
      <xs:extension base="personinfo">

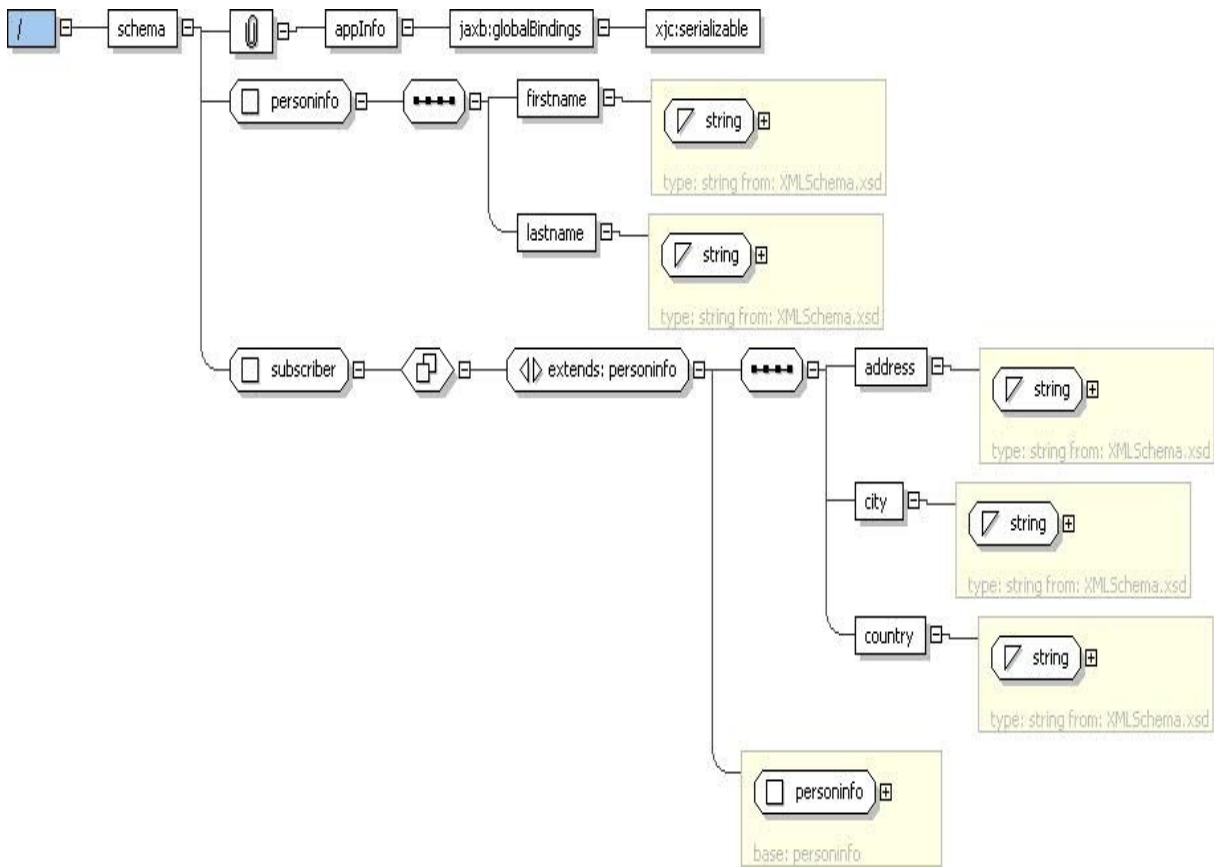
        <xs:sequence>
          <xs:element name="address" type="xs:string"/>
          <xs:element name="city" type="xs:string"/>
          <xs:element name="country" type="xs:string"/>
        </xs:sequence>
      </xs:extension>

    </xs:complexContent>
  </xs:complexType>
</xs:schema>

```

**Table 3 – XSD schema for two types – personinfo and subscriber**





**Figure 4 – Graphical depiction of two XML types – personinfo and subscriber**

```

//
// This file was generated by the JavaTM Architecture for XML Binding(JAXB) Reference
// Implementation, v2.0.4-b01-fcs
// See <a href="http://java.sun.com/xml/jaxb">http://java.sun.com/xml/jaxb</a>
// Any modifications to this file will be lost upon recompilation of the source schema.
// Generated on: 2007.02.08 at 06:27:50 PM GMT
//

package ps;

import java.io.Serializable;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlType;

```

```
/**
 * <p>Java class for personinfo complex type.
 *
 * <p>The following schema fragment specifies the expected content contained within this class.
 *
 * <pre>
 * <complexType name="personinfo">
 *   <complexContent>
 *     <restriction base="{http://www.w3.org/2001/XMLSchema}anyType">
 *       <sequence>
 *         <element name="firstname" type="{http://www.w3.org/2001/XMLSchema}string"/>
 *         <element name="lastname" type="{http://www.w3.org/2001/XMLSchema}string"/>
 *       </sequence>
 *     </restriction>
 *   </complexContent>
 * </complexType>
 * </pre>
 *
 */
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "personinfo", propOrder = {
    "firstname",
    "lastname"
})
public class Personinfo
    implements Serializable
{
    private final static long serialVersionUID = 12343L;
    @XmlElement(required = true)
    protected String firstname;
    @XmlElement(required = true)
    protected String lastname;

    /**
     * Gets the value of the firstname property.
     *
     * @return
     *     possible object is
     *     {@link String }
     *
     */
}
```

```
*/
public String getFirstname() {
    return firstname;
}

/**
 * Sets the value of the firstname property.
 *
 * @param value
 *     allowed object is
 *     {@link String }
 *
 */
public void setFirstname(String value) {
    this.firstname = value;
}

/**
 * Gets the value of the lastname property.
 *
 * @return
 *     possible object is
 *     {@link String }
 *
 */
public String getLastname() {
    return lastname;
}

/**
 * Sets the value of the lastname property.
 *
 * @param value
 *     allowed object is
 *     {@link String }
 *
 */
public void setLastname(String value) {
    this.lastname = value;
}
}
```

**Table 4 – Java class representing personinfo**

```
//  
// This file was generated by the JavaTM Architecture for XML Binding(JAXB) Reference  
Implementation, v2.0.4-b01-fcs  
// See <a href="http://java.sun.com/xml/jaxb">http://java.sun.com/xml/jaxb</a>  
// Any modifications to this file will be lost upon recompilation of the source schema.  
// Generated on: 2007.02.08 at 06:27:50 PM GMT  
//  
  
package ps;  
  
import java.io.Serializable;  
import javax.xml.bind.annotation.XmlAccessType;  
import javax.xml.bind.annotation.XmlAccessorType;  
import javax.xml.bind.annotation.XmlElement;  
import javax.xml.bind.annotation.XmlType;  
  
/**  
 * <p>Java class for subscriber complex type.  
 *  
 * <p>The following schema fragment specifies the expected content contained within this class.  
 *  
 * <pre>  
 * <complexType name="subscriber">  
 *   <complexContent>  
 *     <extension base="{http://www.w3.org/2001/XMLSchema}personinfo">  
 *       <sequence>  
 *         <element name="address" type="{http://www.w3.org/2001/XMLSchema}string"/>  
 *         <element name="city" type="{http://www.w3.org/2001/XMLSchema}string"/>  
 *         <element name="country" type="{http://www.w3.org/2001/XMLSchema}string"/>  
 *       </sequence>  
 *     </extension>  
 *   </complexContent>  
 * </complexType>  
 * </pre>  
 *  
 *  
 */  
@XmlAccessorType(XmlAccessType.FIELD)  
@XmlType(name = "subscriber", propOrder = {
```

```
"address",
    "city",
    "country"
})
public class Subscriber
    extends Personinfo
    implements Serializable
{

    private final static long serialVersionUID = 12343L;
    @XmlElement(required = true)
    protected String address;
    @XmlElement(required = true)
    protected String city;
    @XmlElement(required = true)
    protected String country;

    /**
     * Gets the value of the address property.
     *
     * @return
     *     possible object is
     *     {@link String }
     */
    public String getAddress() {
        return address;
    }

    /**
     * Sets the value of the address property.
     *
     * @param value
     *     allowed object is
     *     {@link String }
     */
    public void setAddress(String value) {
        this.address = value;
    }

    /**
     * Gets the value of the city property.
     *
     */
```

```
* @return
*   possible object is
*   {@link String }
*
*/
public String getCity() {
    return city;
}

/**
* Sets the value of the city property.
*
* @param value
*   allowed object is
*   {@link String }
*
*/
public void setCity(String value) {
    this.city = value;
}

/**
* Gets the value of the country property.
*
* @return
*   possible object is
*   {@link String }
*
*/
public String getCountry() {
    return country;
}

/**
* Sets the value of the country property.
*
* @param value
*   allowed object is
*   {@link String }
*
*/
public void setCountry(String value) {
    this.country = value;
}
```

}
---

**Table 5 – Java class representing subscriber**

### 3.4. Fluent Modelling

In the original form of EC-Assertion, fluents were specified by expressions of the form:

$$\text{valueOf}(\text{fluent\_expression}, \text{value\_expression})$$

whose meaning was that the fluent identified by `fluent_expression` had the value `value_expression` (see Section 2.3). But in the new EC-Assertion language fluents are modelled as relations between objects. More specifically fluents can be defined as relations between objects of the following general form:

$$\text{relation}(\text{Object}_1, \dots, \text{Object}_n)$$

where, *relation* is the name of the relation that takes as arguments *n* variable objects (see Section 3.6.1 for example).

### 3.5. Unification/Relations of objects

In its previous incarnation, where all variables were of basic types, the engine would unify event parameters with formula variables by comparing their types and names, which were both represented as strings.

Currently, DVPv1 represents everything as Java objects internally. Therefore, the unification of event parameters with variables now compares the real Java types of these objects, to make sure that the type of the parameter is either the same as that of the variable or it is a sub-type of the latter.

Similarly, in the initial version of the tool, the equality/inequality relations between variables (or between a variable and a constant) would compare the types of the relation arguments and their values, which were again represented as strings.

Now, the DVPv1 checks that their types are the exact same Java type and, if that comparison was successful, it checks that none of the objects are null. If at least one of them is null then the relation is evaluated to false. Otherwise, the objects' fields need to be compared one by one, recursively. To avoid this recursion, which would need to be implemented with the Java reflection API given that the object types are unknown in general, we have opted to serialise the two objects, that is produce a representation of them as a string of characters, and then compare the two strings for equality (respectively for inequality).

## 3.6. Support for Past-Time Formulae

### 3.6.1. Overview

A past-time formula in EC is a formula that has at least one predicate which occurs before the triggering predicate of the formula. The following formula, for example, is a past-time EC formula:

```

∇ _eID1, _eID2:String; t1, t2:Time
    Happens(e(_eID1, ...), t1,  $\mathfrak{R}(t1, t1)$ ) ⇒
    Happens(e(_eID2, ...), t2,  $\mathfrak{R}(t2, t1)$ )

```

The above formula has a triggering event which is the  $e\_eID1, \dots$  – this event is not time-constrained, so whenever it occurs, it will cause the above formula to be activated. The formula however also has an event  $e\_eID2, \dots$  which should occur at a time point  $t2$  which is *before*  $t1$ . Therefore, this is a past-time formula and the event  $e\_eID2, \dots$  is a past-time event. In LTL for example, we would have to use the past-time operator previously  $P$  to specify the aforementioned formula:  $e\_eID1, \dots \Rightarrow P[e\_eID2, \dots]$ .

Past-time formulae are important for monitoring and especially so for monitoring security and dependability properties, since as we have indicated in [5], the specification of two basic types of such properties in EC, namely confidentiality and integrity, is most naturally done with the use of such formulae. In the following, we give examples of each of these property types.

Example 1: Our first example is based on the smart items scenario that has been identified by the industrial partners of SERENITY in [1]. In this scenario, a patient who had suffered from a cardiac arrest, feels unwell and sends through his EHT a request for assistance to ERC. To establish the cause of the problem, ERC retrieves the patient’s medical record through the EHT. From this record, ERC establishes that the patient’s doctor is on vacation and broadcasts a message to a group of doctors known to be able to substitute the patient’s doctor. A doctor  $D$  receives this message on his own EHT and replies immediately. ERC verifies  $D$ ’s ability to substitute for the patient’s doctor for the specific assistance request. Following this,  $D$ ’s EHT interrogates ERC to receive the patient’s medical data.  $D$  analyses all these data, identifies the most appropriate treatment, and writes the electronic prescription on his/her EHT which subsequently sends the prescription to ERC which forwards it to the patient’s EHT after registering it. In this scenario, Campadello et al. [1] have identified the following confidentiality requirement:

*“A patient’s substitute doctor can access the patient’s medical data if and only if he is the selected doctor” (i.e., Req. 2.2.1.7 in [1])*

Assuming the following operations of ERC,

1. *fetchPatientData(docID:String, request:String, patInfo:MedicalRecord)* – This operation retrieves the medical record of a patient (*patInfo*) given (as input) a medical assistance request associated with the patient (*request*) and the identifier of a requesting doctor (*docID*).
2. *verifyDoctor(docID:String, request:String, verified: Boolean)* – This operation verifies if a doctor (*docID*) can deal with a given request (*request*)

the above requirement can be monitored using the following EC-Assertion rule and assumptions:

### Rule CR1:

```

∀ _eID1,_ercID,_docEhtID,_request:String; _patInfo: MedicalRecord; t1,t2:Time
  Happens (
    e(_eID1,_ercID,_docEhtID, RES-B, fetchPatientData(_docID,_request,_patInfo), _ercID),
    t1,  $\mathfrak{R}(t1,t1)$ )
  ∧ HoldsAt (exposes("fetchPatientData",_docID,_request,_patInfo, _patInfo), t1)
  ⇒ HoldsAt (authorised(_ercID,_docEhtID,
    _ercID,"fetchPatientData"_docID,_request,_patInfo), t1)

```



### Assumption CA1:

**Initially**(exposes("fetchPatientData",\_docEhtID,\_request,\_patInfo))

### Assumption CA2:

```
_eID1, _eID2,_ercID,_docEhtID:String; _verified: Boolean; t:Time
Happens (
    e(_eID2,_ercID,_ercID, RES-A, verifyDoctor(_docID,_request,_verified), _ercID), t,  $\mathfrak{R}(t,t)$ )
    ^ _verified == True
⇒ Initiates(
    e(_eID2,_ercID,_ercID, RES-A, verifyDoctor(_docID,_request,_verified), _ercID),
    authorised(_ercID,_docEhtID,
        _ercID, "fetchPatientData",_docID,_request,_patInfo,) t),
    t)
```

According to *CR1*, following a request for the execution of the operation *fetchPatientData* by a doctor’s EHT to the ERC it should be checked if the requesting doctor’s EHT has been authorised to receive the information that is to be disclosed to him/her. Then, according to *CA2* this authorisation can be obtained through the execution of *verifyDoctor*. Finally, *CA1* specifies that the operation *fetchPatientData* discloses *patInfo*.

*CR1* is a past-time EC formula since the truth value of the predicate

```
HoldsAt (authorised(_ercID,_docEhtID,
    _ercID, "fetchPatientData",_docID,_request,_patInfo)), t1)
```

in the formula should be established at the same time as the truth value of the unconstrained predicate

```
Happens ( e(_eID1,_ercID,_docEhtID, RES-B, fetchPatientData(_docID,_request,_patInfo),_ercID),
    t1,  $\mathfrak{R}(t1,t1)$ )
```

To see why this is a past-time formula, it suffices to consider the definition of the *HoldsAt* predicate, according to the rules (EC1)-(EC3) of Table 1 on page 9. As we can see there, the *HoldsAt* predicate effectively demands to find predicates *Initially* or *Initiates* in the past which have not been followed by a *Terminates* predicate (again in the past with respect to the current moment).

**Example 2:** In the smart items scenario of SERENITY, the following integrity requirement has also been identified:

*“Electronic prescriptions shall be issued only by doctors by means of an e-health terminal.” (i.e., Req. 2.2.1.15 in [1])*

This requirement can be monitored by a rule stating that if an ERC receives an electronic prescription by a doctor then this doctor must be authorised to issue the prescription. The rule can be created by instantiating the destination party integrity monitoring pattern assuming that:

1. ERC provides an operation to create new electronic prescriptions (*presc*) for a medical assistance request (*request*):

*createPrescription(docID:String, request:String, presc: Prescription)*

2. Doctors are authorised through the execution of the operation *verifyDoctor* of ERC.

The above requirement can be monitored at runtime using the following EC rule and assumptions as we have discussed in :

### Rule IR1:

```

∇ _eID1,_ercID,_docEhtID:String; t:Time
  Happens (
    e(_eID1,_docEhtID,_ercID,REQ-B,createPrescription(_docID,_request,_presc), _ercID),
    t,  $\mathfrak{R}(t,t)$ )
  ∧ HoldsAt (transforms("createPrescription",_docID,_ request,_presc, _ercID), t)
  ⇒ HoldsAt (authorised(_ercID, _docID, _ercID,
    "createPrescription",_docID,_ request,_presc)), t)

```

### Assumption IA1:

```
Initially (transforms("createPrescription", _docID,_ request,_presc, _ercID))
```

### Assumption IA2:

```

∇ _eID2,_ercID,_docEhtID:String; t:Time;
  _request: String, _verified: Boolean
  Happens (
    e(_eID2,_ercID,_ercID, RES-A, verifyDoctor(_docID, _request,_verified),_ercID), t, $\mathfrak{R}(t,t)$ )
  ∧ _verified == True
  ⇒ Initiates (e(_eID2,_ercID,_ercID, RES-A, verifyDoctor(_docID,_request,_verified), _ercID),
    authorised(_ercID, _docEhtID, _ercID,createPrescription",_docID,_ request,_presc)), t)

```

The rule *IR1* checks whether a doctor (*\_docID*) who invokes the operation *createPrescription* in ERC (*\_ercID*) is authorised to do so. It is interesting to note, due to the mapping of the pattern variables of Table 2, *IR1* effectively describes a delegation of the doctor’s right to create prescriptions to his/her EHT, since it is the latter which is the sender in this interaction (*\_docEhtID*), while it is the doctor who is being authorised (*\_docID*) for the action in reality.

The assumption *IA2* above states that a doctor is authorised to call the operation *createPrescription* in ERC only if it is verified by the operation *verifyDoctor*. In this case, an appropriate authorisation fluent will be generated by *IA2* and by virtue of the EC axioms shown in Table 1 we can derive that the *HoldsAt* predicate in the head of the rule *IR1* is satisfied.

### 3.6.2. Method for Checking Past-Time Predicates

The method for checking past-time predicates works in the following manner – if while observing the trace of events in the system we obtain an event which could be unified with a part of a formula instantiation but its time constraints cannot yet be evaluated (because it is constrained by a future event), then we store this event in the database of past-time events, which is depicted as “Database

I' in Figure 2. If later on we observe an event which can be unified with that formula instantiation and which fixes some of the up to that point unspecified time instances mentioned in the formula, then we search in the database of past-time events for any event which could currently be unified.

### 3.7. Support for HoldsAt Predicates

A similar problem with the past-time formulae exists for the HoldsAt predicate, since its definition is based on the existence of past-time Initiates and Terminates predicates. These predicates are derived from the assumptions of a theory which state what events initiate and terminate a particular fluent respectively.

#### 3.7.1. Fluent Initiates and Terminates Database

In order to be able to support the HoldsAt predicate we need to store the predicates Initiates and Terminates when these are derived, for use at a later time in the evaluation of future HoldsAt predicates, just like we did for the support of past-time formulae. We therefore store the predicates Initiates and Terminates into the database depicted as “Database II” in Figure 2. When later on we need to evaluate a HoldsAt at some future time instance  $t_1$ , we search this database for the most recent Initiates and Terminates predicates which precede  $t_1$  and we use the EC axioms shown in Table 1 to derive the truth value of the HoldsAt predicate in the head of the rule *IRI* is satisfied.

#### 3.7.2. Query Generation for HoldsAt

The truth value of *HoldsAt* predicates in formulae is established by querying the fluents initiation and termination database that is maintained by the engine. The query to determine this status is derived from the axioms of Event Calculus. More specifically, given the following axioms for the *HoldsAt* predicate:

$$EC1: \text{Clipped}(t_1, f, t_2) \Leftarrow (\exists e, t) \text{Happens}(e, t, \mathfrak{R}(t_1, t_2)) \wedge \text{Terminates}(e, f, t)$$

$$EC2: \text{HoldsAt}(f, t) \Leftarrow \text{Initially}(f) \wedge \neg \text{Clipped}(0, f, t)$$

$$EC3: \text{HoldsAt}(f, t_2) \Leftarrow (\exists e, t) \text{Happens}(e, t, \mathfrak{R}(t_1, t_2)) \wedge \text{Initiates}(e, f, t) \wedge \neg \text{Clipped}(t, f, t_2)$$

We can derive the following formula for *HoldsAt* by taking the disjunction of EC3 and EC4 and substituting EC1 for the predicate *Clipped* in the formula:

$$(Q1) \quad \text{HoldsAt}(f, t_2) \Leftarrow$$

$$\quad (\text{Initially}(f)$$

$$\quad \wedge \neg \exists e_1, t_3: (\text{Happens}(e_1, t_3) \wedge (0 < t_2) \wedge (t_3 < t_2) \wedge \text{Terminates}(e_1, f, t_3)))$$

$$\quad \vee (\text{Happens}(e_2, t_1) \wedge \text{Initiates}(e_2, f, t_1) \wedge (t_1 < t_2)$$

$$\quad \wedge \neg \exists e_3, t_4: (\text{Happens}(e_3, t_4) \wedge (t_1 < t_4) \wedge (t_4 < t_2) \wedge \text{Terminates}(e_3, f, t_4)))$$

The formula (Q1) constitutes the basis of querying the status of the fluent initiation and termination database of the monitoring engine to establish the truth value of *HoldsAt* predicates.

## 4. Limitations

---

In this section we will discuss a number of limitations of DVPv1. We expect to address most of these in the second version of the prototype, which is to be delivered in the deliverable A4.D3.3.

### 4.1. Support for Future-Time HoldsAt Predicates

It is always possible that one specifies a formula where a HoldsAt predicate needs to be evaluated in the future, for example,  $\text{Happens}(e, t1) \Rightarrow \text{HoldsAt}(f, t2) \wedge t1 \leq t2$ . In the current state of the prototype, the query for the aforementioned formula will be performed in the time instance  $t1$  and therefore the truth value of the HoldsAt predicate will be computed with respect to the Initiates and Terminates predicates which have occurred up to the time instance  $t1$ .

It is evidently the case that this way of treating future-time HoldsAt predicates is incorrect, since there can be more Initiates and Terminates predicates between the current time instance ( $t1$ ) and the future time instance ( $t2$ ) at which we are interested in computing the value of the HoldsAt predicate.

This limitation will be corrected in the second version of the prototype.

### 4.2. Lack of Multiple Inheritance

The object types which can be defined by the user can unfortunately only make use of single inheritance, not multiple one. This is a direct consequence of the fact that the XML schema does not allow one to describe a data structure which extends (i.e., inherits from) more than one data structure.

As such, if multiple inheritance is needed, say type  $Z$  to inherit from types  $A$ ,  $B$  &  $C$ , then one will have to simulate it somehow, either by declaring that  $Z$  inherits from  $A$  and contains sub-objects/fields of type  $B$  &  $C$ , or by declaring three different types:  $ZA$  which inherits from  $A$ ,  $ZB$  which inherits from  $B$  and  $ZC$  which inherits from  $C$  and making as many copies of the formulae which should contain a  $Z$  object to cover all cases. Unfortunately, none of these workarounds are universally applicable. We do not currently plan to attack this limitation due to the fact that we do not wish to abandon the representation of events and the EC language through XML.

### 4.3. Support for Logical Clocks

As it is, DVPv1 assumes the existence of a global clock which is used to timestamp all the events it receives. It also assumes that the events received are received in order.

Clearly, these assumptions do not hold in a distributed system, since the local clocks of the various event collectors cannot be easily synchronised in such a setting. Therefore, it would be desirable to extend the DVP so that it can deal with a vector of clocks, similarly to the concept of logical clocks. We will investigate this possibility for DVPv2 but there is also the possibility that this will require considerable support from the general SERENITY Framework and the event collectors.

To see why, consider the case where we are capturing events at a component both at the SOAP level and at the system call level. Then the logical clocks reported by these two captors will need to be related – each component will have to keep a single representation of a (local) logical clock, containing the clocks of all the other components it is communicating with. Then all event collectors will need to use the value of that logical clock when reporting events.

#### 4.4. Support for Non-Determinism

DVPv1 has no support for non-determinism – all its fluents have a specific value always. Nevertheless, it would be interesting to examine the possibility of offering support for non-determinism, so that we can describe systems whose internal behaviour is not known. To do so we will need to provide support for the `Releases` predicate of EC. We will also have to examine how this predicate behaves in the presence of our complex, object-based fluents.

#### 4.5. Support for a Full Object-Oriented Database

The databases used in DVPv1 for storing past-time events and `Initiates/Terminates` predicates are currently simple, in-memory database. We are planning to replace that with a full-fledged Object-Oriented database, namely `db4objects`<sup>1</sup>.

#### 4.6. Minimisation of Past-Time Predicates in the DB

DVPv1 stores all past-time predicates in the DB and never deletes them from it. Given that the system may run forever, this will eventually lead to a situation where the DB is full and no more predicates can be stored in it. We will therefore examine ways to minimise the number of past-time predicates held at each time instance in the database.

#### 4.7. Calling External Functions

In order to be able to support control and recovery, DVPv2 will need to be able to perform calls to external functions, so as to actively poll for specific events at particular instances, to be able to respond to the events it receives for informing the SERENITY Framework of what the correct action should be at the particular circumstances. However, DVPv1 can only call a restricted set of internal functions, used mainly for keeping statistical information.

#### 4.8. Supporting Internal Alarms

Again, in order to be able to support control and recovery, we will need to provide support for internal alarms and alarm handlers in the DVP. This will allow it to poll for a particular event at regular intervals, to perform some control action at specific time instances, etc.

---

<sup>1</sup> <http://www.db4objects.com/community/>

## 5. Installation and Usage Guide

---

### 5.1. Required Software

To use the city monitoring tool, the user should download and install on his/her machine:

- Version 5.0.14 of the Tomcat server – This server can be downloaded from <http://tomcat.apache.org/>. An installation guide for the server is also available at the same site. Please consult the release note of tomcat for the selection of right XML parser.
- Version 1.4 of Axis server – This server can be downloaded from <http://ws.apache.org/axis/>. An installation guide for the server is also available at the same site.

### 5.2. Installation

- To install the monitoring manager, extract the files in the archive into the folder `C:\Monitor`
- To install the data analyzer copy the folder `C:\Monitor\analyzer\code` in the `classes` folder of the axis installation in Tomcat. This prototype assumes that the Tomcat server is deployed on port number 8080 (i.e. default port for tomcat) .

### 5.3. How to Use the DVP

#### 5.3.1. The Data Analyzer

To start the data analyzer, the user has to start the tomcat server by executing the startup file in the `TOMCAT_HOME\bin` folder.

- To use the analyzer with the City monitoring manager, in a command prompt window give the command `C:\Monitor\analyzer\deploy` The data analyzer is up and the wsdl specification of the data analyzer service can be seen at:

<http://localhost:8080/axis/services/analyzerService?wsdl>

and the analyzer service endpoint is

<http://localhost:8080/axis/services/analyzerService>

Section 5.3.2 describes how to use City monitoring manager and data collector.

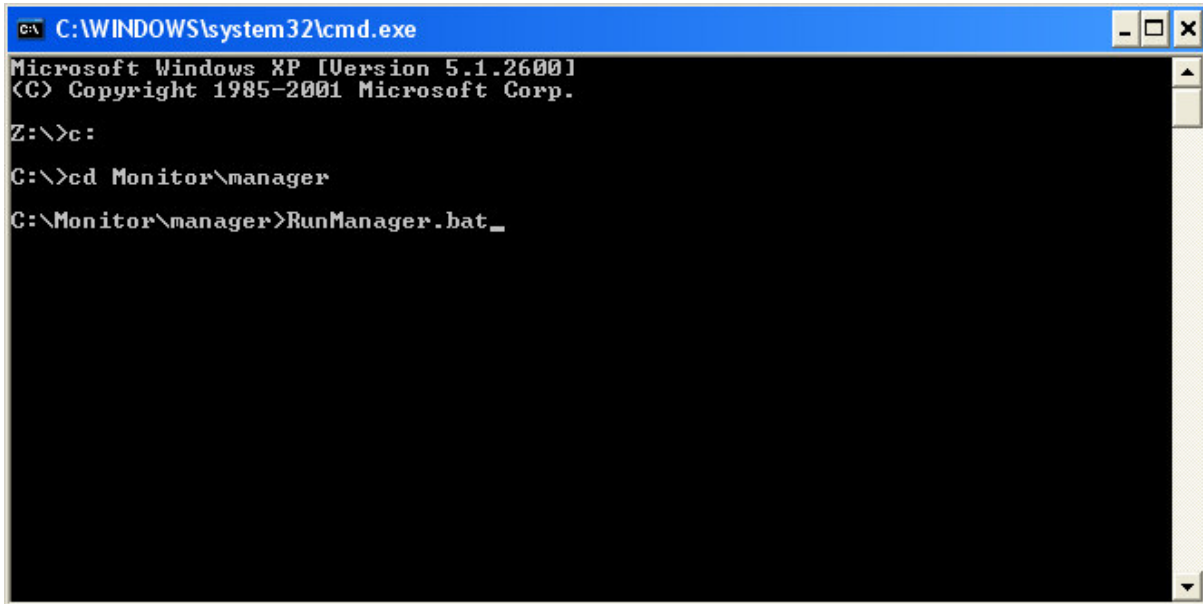
#### 5.3.2. The Monitoring Manager

The monitoring manager is used to import and select the formulae to be monitored, send the selected formulae to the data analyzer, start the data collector for a monitoring session, initiate a polling process that retrieves possible violations of the properties and view the result of monitoring. To retrieve violations of properties, the monitoring manager polls the data analyzer at regular time intervals that can be specified by the user and shows the results that

it retrieves in a formula viewer.

To use the monitor manager, follow the following steps:

1. To start the monitor manager, in a command prompt window execute the command `C:\Monitor\manager\RunManager` as shown in the figure below. Following this, the monitor manager window will pop up.

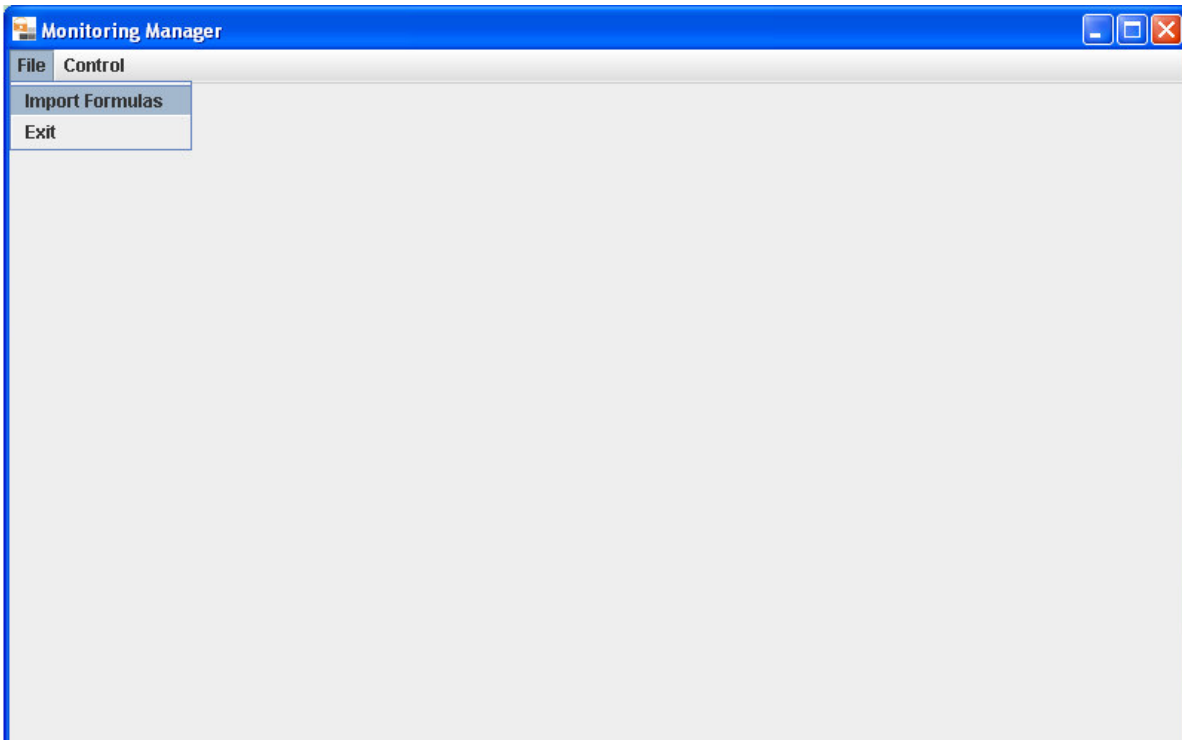


```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

Z:\>c:
C:\>cd Monitor\manager
C:\Monitor\manager>RunManager.bat_
```

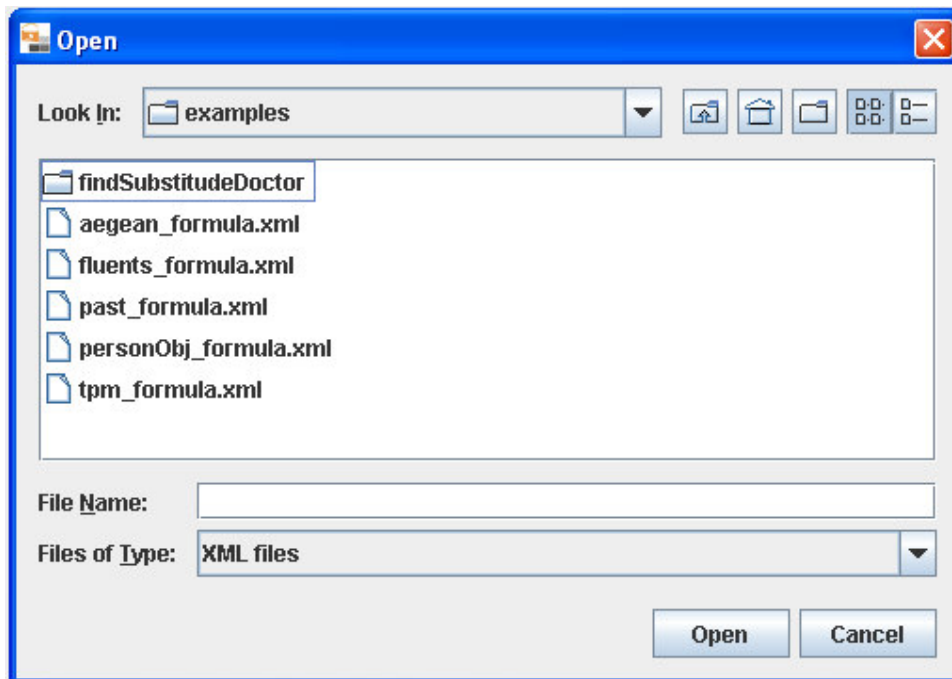
**Figure 5 – Command prompt window**

2. Then, to import the formulae to be monitored, select the option "Import Formulae" from the "File" menu of the manager, as shown in the figure below.



**Figure 6 – How to import the formulas in the Monitoring Manager**

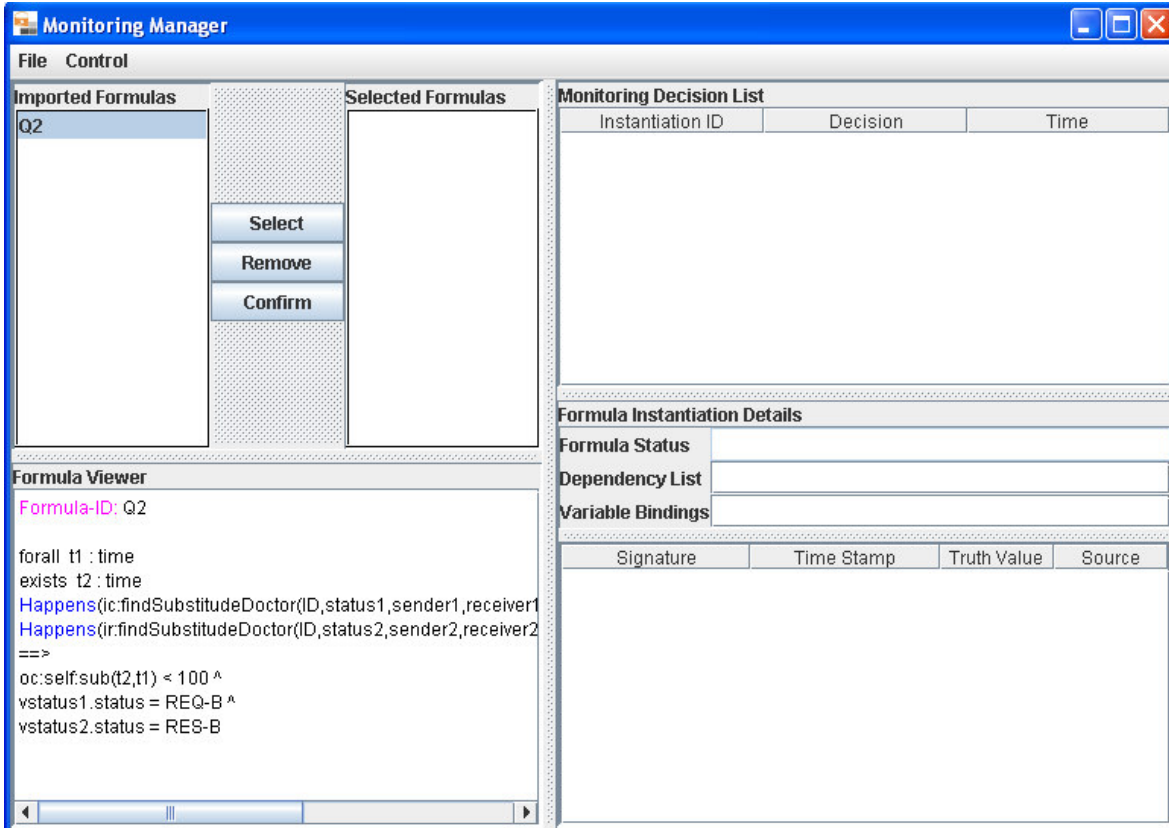
3. In the file opening dialog box that appears following, choose the XML file that contains the formulae that you want to monitor.





**Figure 7 – The file opening dialogue box**

4. The monitor manager will then read all the formulae from the file and display the formulae in the display panel of the monitoring manager as illustrated in Figure 8.



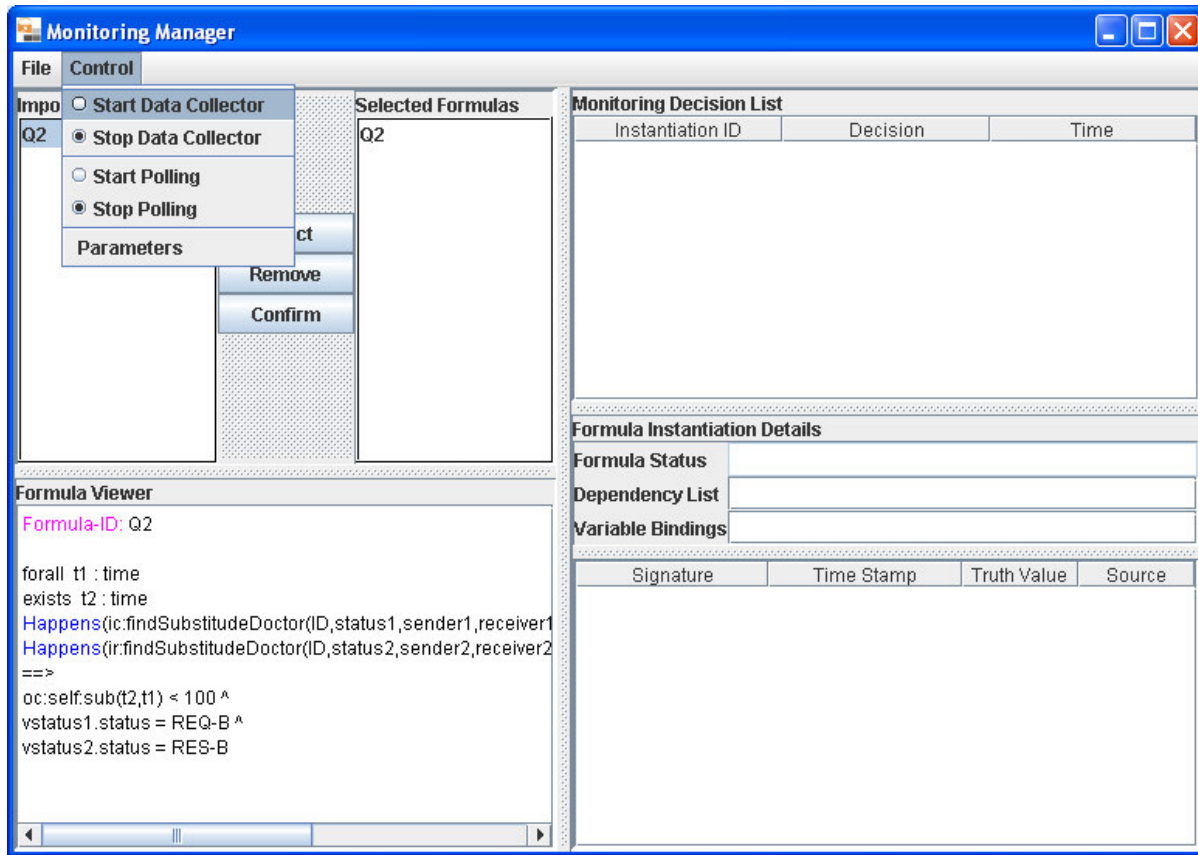
**Figure 8 – The Monitoring Manager**

The monitoring manager lists the identifiers of the imported formulae in the "Imported Formulae" panel. To view a formula in the event calculus format, the user may select its ID. Following this selection, the formula with the selected ID will be shown in the "Formula Viewer" panel of the manager. If the user wants to select the formula to be monitored, he/she may select its ID in the imported formulae panel and click on the "Select" button. Following this, the selected formula will appear in the "Selected Formulae" panel. The user may repeat the same process to select more formulae. When the selection is complete, the user can click on the "Confirmed" button, to send the formulae to the data analyser. If the submission of formulae to the analyser is successful, the monitor manager will show the following message. The user should press the "Ok" button to continue.



**Figure 9 - Information dialogue box**

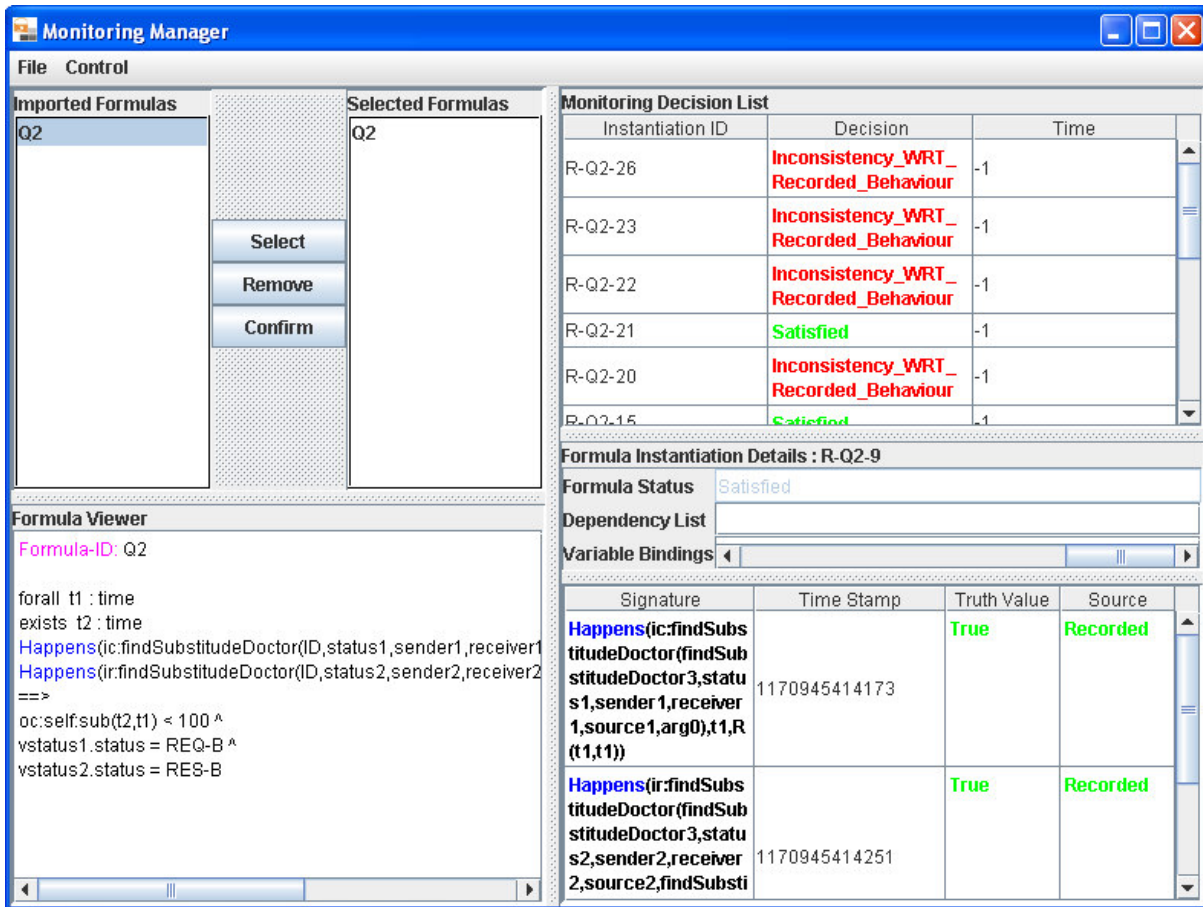
5. The next step is to provide the analyzer with runtime events. From the Start Data Collection command in the Controls menu of the monitor manager we can activate the collection of the events. The monitor manager can by default accept events in the port number 12345 and report them to the analyzer. The format of those events is based in the event XML schema described in [1]. Any event collection mechanism that can provide events according to the defined XML schema can be used to report the events. For the purposes of the demo we can use the SOAP event collector which can be located in `C:\Monitor\SoapCollector`. This application creates a proxy service which listens to a user defined port number, accepts incoming SOAP messages, translates them to a Serenity event format and then forwards them to the real web service for the execution of the service and to the monitor management tool. For the execution of the collector the user must type `java -cp xercesImpl.jar code.TcpTunnel` followed by the parameters of the listening port, the IP address and port of the real web service and the IP address and port of the monitoring manager. In our case, where the Tomcat is deployed locally in the default port (i.e. 8080) and the manager listens for events in the 12345 port, the execution command should be `java -cp xercesImpl.jar code.TcpTunnel 8081 localhost 8080 localhost 12345`. Now that the collector is activated we must inform the monitor manager to accept any events are send to it by selecting the Start Data Collector from the Control menu. Any attempt to invoke a web service in the port 8081 will result to report this event to the monitor manager tool.



**Figure 10 – How to stop data collector in the Monitoring Manager**

6. The user may stop the data collector by selecting the option "Stop Data Collector" in the "Control" menu
7. To start polling the data analyser in order to view the violations of the formulae being monitored, the user should select the option "start polling" from the control menu. Following this, the monitor manager will start polling the data analyser at regular time intervals (the default time interval is 10 seconds).

The monitoring manager shows the list of instances of the violated and satisfied formulae in the "Monitoring Decision List" panel as shown in Figure 11. This panel will be updated at the regular intervals. The Monitoring Decision List will show the monitoring summary of each instance of each formula. The left most column in this list shows the unique formula instance ID, the middle column shows the decision for the formula instance, and the right most column shows the time when the decision was made by the data analyzer.



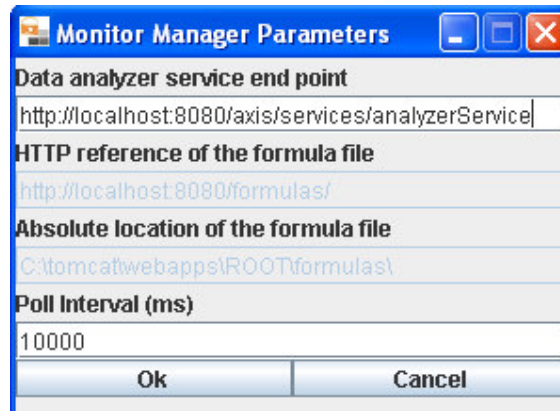
Instantiation ID	Decision	Time
R-Q2-26	Inconsistency_WRT_Recorded_Behaviour	-1
R-Q2-23	Inconsistency_WRT_Recorded_Behaviour	-1
R-Q2-22	Inconsistency_WRT_Recorded_Behaviour	-1
R-Q2-21	Satisfied	-1
R-Q2-20	Inconsistency_WRT_Recorded_Behaviour	-1
R-Q2-15	Satisfied	-1

Signature	Time Stamp	Truth Value	Source
Happens(ic.findSubstituteDoctor(findSubstituteDoctor3,status1,sender1,receiver1,source1,arg0),t1,R(t1,t1))	1170945414173	True	Recorded
Happens(ir.findSubstituteDoctor(findSubstituteDoctor3,status2,sender2,receiver2,source2,findSubsti	1170945414251	True	Recorded

**Figure 11 – The Monitoring Manager showing a list of violations detected**

To view the details of a formula instance, the user should select the relevant formula instance in the Monitoring Decision List. Following this, the monitor manager shows the details of the formula instance in the "Formula Instantiation Details" panel. This panel displays the formula status, other formulae that the specific formula may depend on (see [4] for a definition of formula dependencies and how they are used in monitoring), and the values bound to the variables of the formula. "Formula Instantiation Details" panel also shows the truth values of the individual predicates of the formula, the timestamps of the establishment of these truth values, and the source of the information that underpins them ("Recorded" for events generated directly by the system under observation and "Derived" for events generated by deductive reasoning as described in [4]).

- To change the polling interval, the user should select the option "stop polling" in the main monitor manager window, then select "parameters" from the "control" menu in this window and, in the dialog box that pops up, specify a new value for the polling interval. Subsequently, the user should select the option "restart polling" from the control window.



**Figure 12 – Monitor Manager parameters dialoge box**

9. To stop the manager, first stop the data collector (if it is running), stop polling, then select exit from file menu.

## 5.4. Example of Using the DVP Together With an Event Collector

In this section we will demonstrate a simple example of the usage of the analyzer and the monitoring tool. To make use of this example you must follow the following steps.

- Install Tomcat and Axis.
- Start Tomcat using the default port number(i.e. 8080).
- Deploy the analyzer service as described in 5.2 .
- In the same way as in the previous step deploy the service located in `C:\Monitor\examples\findSubstituteDoctor`, only this time copy the `dri` folder in the `classes` folder of axis.
- Check that both `analyzerService` and `DoctorRegistryInquiryPT` web services have been deployed correctly by pointing your browser at `http://localhost:8080/axis/servlet/AxisServlet`.
- Start the monitor manager.
- Import the formulae located in the `aegean_formula.xml` file in the `C:\Monitor\examples` folder. Select the Q2 formulae and press Confirm to report the selected formulae to the analyzer service.
- Start the SOAP Collector by executing the `RunCollector.bat` file from the `C:\Monitor\SoapCollector` folder.
- Activate the Data Collection and the Polling from the Controls menu of the monitor manager.
- Execute the `RunTestDRI.bat` file from the `C:\Monitor\examples\findSubstituteDoctor` folder which invokes the `DoctorRegistryInquiryPT` web service.

## 6. Conclusion

---

This document has presented the version of the Dynamic Validation Prototype (DVP) which had been developed at City University prior to the SERENITY project and then described how the new version of the prototype which we have delivered as part of deliverable A4.D3.1 differs from it. It has presented the architectural differences between the two prototypes as well as all the extensions which have been developed for the DVP, namely:

- New EC Assertion Language
- New structure of events
- Support for complex object types in the EC Assertion Language
- Support for user-defined complex object types
- Support for inheritance in object types
- Modelling of complex fluents
- Support for past-time EC formulae
- Support for HoldsAt predicates

The document has also identified a number of limitations of the current version of the DVP, namely:

- Support for multiple inheritance in types
- Support for future-time HoldsAt predicates
- Support for logical clocks
- Support for non-determinism
- Support for a full object-oriented database
- Minimisation of past-time predicates stored at any time
- Ability to call external functions
- Support for internal alarms

Finally, this report contains a guide for the installation and usage of the software which has been delivered as part of A4.D3.1.

## References

- [1] Campadello S., Compagna L., Gidoïn D., Giorgini P., Holtmanns S., Latanicki J., Meduri V., Pazzaglia J.-C., Seguran M., Thomas R., Zanone N. (2006): S&D Requirements specification, Deliverable A7.D2.1, SERENITY Project
- [2] Kloukinas C., Ballas C., Presenza D., and Spanoudakis G. (2006): Basic set of Information Collection Mechanisms for Run-Time S&D Monitoring, Deliverable A4.D2.2, SERENITY Project
- [3] Mahbub K. (2006): Runtime Monitoring of Service Based Systems, Ph.D. Thesis, Department of Computing, School of Informatics, City University, London, U.K.
- [4] Mahbub K., Spanoudakis G. (2004): A Framework for Requirements Monitoring of Service Based Systems , 2nd International Conference on Service Oriented Computing, November, New York, U.S.A.
- [5] Shanahan, M. P. (1999): The Event Calculus Explained, in Artificial Intelligence Today, LNAI no. 1600:409-430, Springer
- [6] Spanoudakis G., Kloukinas C., Androutopoulos K. (2006): Towards Security Monitoring Patterns, 22<sup>nd</sup> Annual ACM Symposium on Applied Computing, Technical Track on Software Verification (to appear)
- [7] Spanoudakis G., Mahbub K. (2006): Non Intrusive Monitoring of Service Based Systems, International Journal of Cooperative Information Systems, Vol. 15, No. 3, pages 325-358