A4.D3.3 – V2 of Dynamic Validation Prototype

K. Mahbub, G. Spanoudakis, C. Kloukinas

Document Number	A4.D3.3
Document Title	V2 of Dynamic Validation Prototype
Version	1.0
Status	Final
Work Package	WP 4.3
Deliverable Type	Prototype
Contractual Date of Delivery	30 June 2007
Actual Date of Delivery	10 September 2007
Responsible Unit	CUL
Contributors	CUL
Keyword List	S&D Monitoring Tool
Dissemination level	PU

Change History

Version	Date	Status	Author (Unit)	Description
0.1	20/7/2007	Draft	K. Mahbub (CUL)	Table of contents, indicative section contents
0.2	25/7/2007	Draft	C. Kloukinas (CUL)	New TOC, Sections 2 & 4
0.3	31/7/2007	Draft	K. Mahbub (CUL)	Section 3.2, 4
0.4	10/8/2007	Draft	G. Spanoudakis (CUL)	Editing, Algorithm Specification, Examples, Change of TOC
0.5	15/8/2007	Draft for Quality Approval	G. Spanoudakis (CUL)	Final version submitted for quality approval
1.0	9/9/2007	Final	G. Spanoudakis (CUL)	Final version

Executive Summary

This report is part of the deliverable A3.D3.3 and provides a description of the second version of the Dynamic Validation Prototype that has been developed in SERENITY. The code that implements the 2nd version of this prototype is also part of A3.D3.3. The report describes the extensions of the first version of the Dynamic Validation Prototype which has been implemented in version 2, and other amendments of the original prototype that were implemented in the second version.

In summary, the implemented extensions enable the monitoring of rules which refer to events that have been captured by event collectors running on different machines from the monitoring engine and, possibly, from each other. The main problem with such events is that they do not have comparable timestamps. To overcome this problem, the second version of the prototype provides a framework for synchronising the clocks of the machines where events are captured with the clock of the monitor and implements a new version of the monitoring algorithm which takes into account possible delays in the transmission of events over a network before applying the principle of negation as failure in order to decide about the truth value of a predicate in a formula. Furthermore, the new version of the prototype incorporates a garbage collector which deletes events from the event memory of the prototype when these events can no longer be needed for the reasoning process that is implemented by the monitor.

Other changes that have been implemented in the second version of the dynamic validation prototype are related to pruning of monitoring formula templates during the monitoring process. Possible cases of pruning were identified during the implementation of the initial version of DVP and their implementation led to significant improvements in performance.

Table of Contents

1. Introduction

This report is part of the SERENITY A4.D3.3 deliverable and its purpose is to describe the implementation of the 2nd version of the dynamic validation prototype of SERENITY and give guidelines on how to install and use it. In addition to this report, A4.D3.13 includes:

— the source code of the 2nd 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 collectors1 that need to be installed in order to generate the events required for monitoring the formulae described in (2).

The second version of the dynamic validation prototype, shortly referred to as *DVPv2* in the rest of this document, extends the first version of the prototype by introducing mechanisms that enable monitoring in the presence of events which are captured by distributed event collectors and transmitted to the monitor over a network. It also introduces some optimisations related to the pruning of monitoring formula templates during the reasoning process. These extensions were all necessary in order to widen the monitoring capability of the prototype and improve its performance.

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 DVPv2, provide an overview of limitations of the first implementation of DVP and the actions that were taken to address them (if any) as part of DVPv2, and outline plans for future work on it. In Section 4, we provide guidelines for the installation and use of DVPv2. Finally, in Section 5, we provide some concluding remarks for DVPv2.

[1] In this report, we use the terms "event collector" and "data collector" interchangeably to refer to the components that capture the runtime events (data) used by the monitor.

2. Overview of DVPv1

DVPv1 has been developed in order to provide support for:

— The communication with the event collectors 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 [4])

— 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 [9] can be expressed by past-time EC-Assertion formulae (see also [1] for a summary of EC-Assertion).

The architecture of DVPv1 is illustrated in Figure 1. 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 system to be monitored 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 system to be monitored 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 event buffer is a subcomponent of the monitor manager because the event collector is no longer limited to receiving events just web services. Instead, the event collector in SERENITY can receive events from more general types of systems via TCP/IP sockets as well as from web services.

The *monitor* retrieves the events which are recorded in the event buffer during the operation of the system that is being monitored in the order of their occurrence, derives other possible events that may have happened without being recorded (based on assumptions set for the 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 system that is being monitored.

Figure 1 also shows some of the internal components of the monitor. 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.

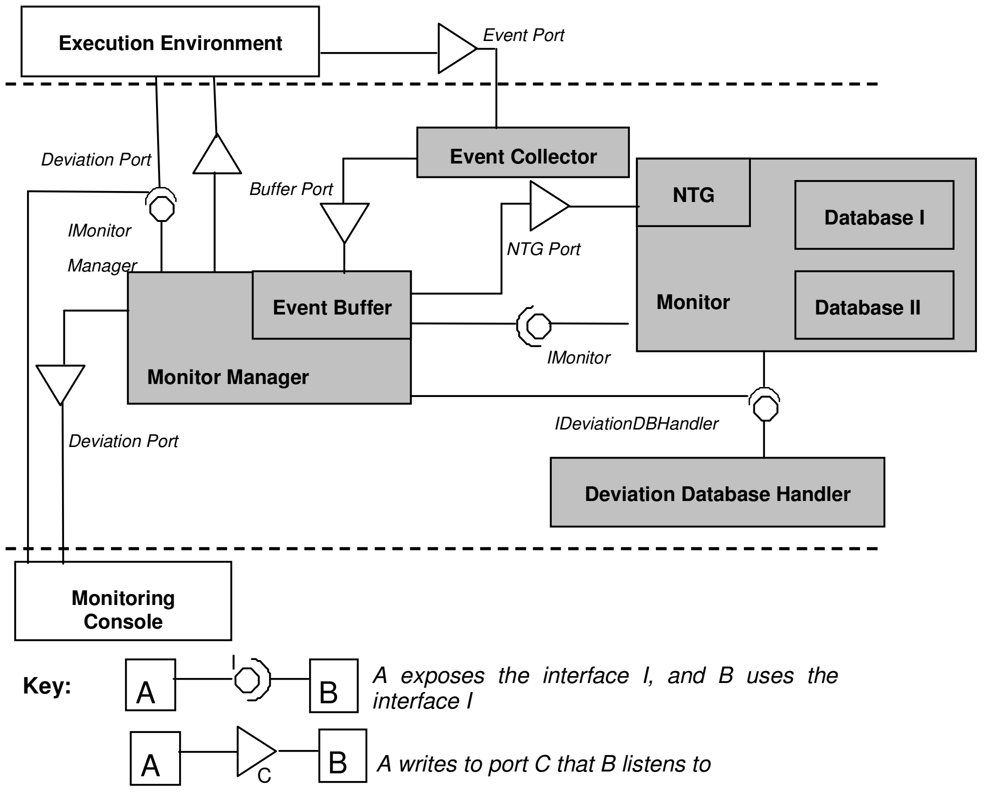


Figure 1 – Architecture of DVPv1

3. DVPv2

3.1. Overview of extensions

In summary, the implemented extensions enable the monitoring of rules which refer to events that have been captured by event collectors running on different machines from the monitoring engine and, possibly, from each other. The main problem with such events is that they do not have comparable timestamps. To overcome this problem, the second version of the prototype provides a framework for synchronising the clocks of the machines where events are captured with the clock of the monitor and implements a new version of the monitoring algorithm which deals with the possibility of getting events not in the order that they have been produced. Furthermore, the new version of the prototype incorporates a garbage collector which deletes events from the event memory of the prototype when these events can no longer be needed for the reasoning process that is implemented by the monitor.

Other changes in the second version of the dynamic validation prototype are related to pruning of monitoring formula templates during the monitoring process. Possible cases of pruning were identified during the implementation of the initial version of DVP and their implementation led to significant improvements in performance.

3.2. Handling events captured by distributed event collectors

In Serenity, monitoring may be based on events which are captured from event collectors attached to components that implement security solutions (aka "implementations" in [6]) or other functionality which may be running on different machines. This creates a problem as the timestamps of events captured from distributed sources are generated by different system clocks which are not synchronised and, as a result, they might not be directly comparable. Thus, a transformation of the time stamps of the different events onto a common timeline is necessary in order to enable a sound checking of the time conditions that are expressed in monitoring rules. As an example consider the following monitoring rule:

```
Rule 1:
∀ _eID1,_ercID,_docEhtID,_request:String; _patInfo: MedicalRecord; t1,t2:Time
Happens( e(_eID1,_ercID,_docEhtID,RES-B,
         fetchPatientData(_docID,_request,_patInfo),_ercID), t1, ℜ(t1,t1)) ⇒
Happens( e(_eID2,_ercID,_ttpasID, RES-A,
         verifyDoctor(_docID,_request,_verified), _ttpasID), t2, ℜ(t2,t1))
         ∧ (_verified == True)
```

This rule can be used to monitor the following confidentiality requirement that has been identified in [2]:

> *"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 [2])*

The rule is relevant to the smart items scenario of SERENITY and specifies that before the emergency response centre (`_ercID`) responds to an invocation of the operation *fetchPatientData* which is invoked by the e-health terminal of a doctor (`_docEhtID`) to retrieve the medical record of a patient (`_patInfo`), it must have called the operation *verifyDoctor* in trusted third party that provides a doctor authentication service (`_ttpasID`) and received a positive verification of the identifier of the enquiring doctor (`_docID`).[2] The rule expresses this condition by stating that the timestamp `t2` of the event `e(_eID2,_ercID,_TTP,RES-A,verifyDoctor(_docID,_request,_verified), _TTPAS)` must be less than the timestamp `t1` of the event that indicates the response to the *fetchPatientData* invocation. As in this case, however, `t1` refers to the time of emergency response centre and `t2` refers to the time of the trusted authentication service the two values are not directly comparable unless the time difference between the clocks of the machines where these two components of the system run is known. Furthermore, since the relevant events may have been transmitted to the monitor over different networks with different communication delays it is also uncertain how soon after the occurrence of the events at their sources the monitor will receive them.

To deal with such cases it was necessary to extend:

- event collectors in order to enable a synchronisation of their clocks and ensure that the timestamps of the events that they generate are expressed on the same timeline and
- the monitoring engine in order to ensure that it considers the events which are required by the rules even if they arrive well after they occur and not in the order of their occurrence at source.

In the following, we describe the above extensions.

3.2.1. Extension for Synchronisation of Collector and Monitor Clocks

To enable the synchronisation of event timestamps, we have extended event collectors and the monitor with components that realise the *Network Time Protocol* (NTP) [7]. The implementation of this protocol in *DVPv2* allows event collectors to compute the difference of their clocks with the monitor clock at regular intervals. This difference is used to transform timestamps taken according to the clock of each collector to timestamps that express time in terms of the monitor's clock. This is achieved by implementing an *NTP client* at each event collector and an *NTP server* at the machine that hosts the monitoring manager and monitor, as shown in Figure 2. NTP clients attached to the collectors are calling the NTP server at regular intervals to synchronize their clocks with the clock of server at a regular intervals.

More specifically, an NTP client synchronizes its clock with an NTP server's clock by exchanging several packets with that server. The NTP client stores its own time stamp in each packet that it sends to the server. When the server receives the packet, it will store its own timestamp and a transmit timestamp in it and sends it back to the client. Upon the receipt of the reply packet from the server, the client logs the receipt time of the packet in order to estimate its travelling time. The NTP client uses these time differences to estimate the time offset between both machines, as well as the standard deviation of the time difference.

In our solution we assume that the machine hosting the monitor and the monitoring manager is running an NTP server. Most of the operating systems, including Windows and Linux, incorporate

[2] This rule is a simplification of a similar rule that we specified in [9] that does not use assumptions and fluents.

built in NTP servers/daemons as part of time services[3]. Our implementation is based on Windows and, therefore, accesses the NTP server of the Windows time service. The NTP client has been implemented in Java.

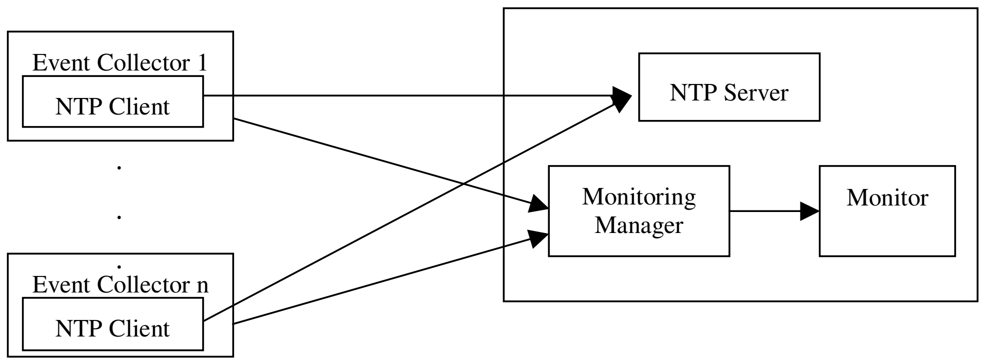


Figure 2 – Event Serialization Architecture in DVPv2

3.2.2. Extensions to the Monitor Engine

The extension of event collectors and the monitor engine which computes the difference of the clock of each collector with respect to the monitor clock enables the ordering the different events according to their timestamps but it does not solve the second problem that we pointed out above, that is the problem which arises from the fact that even if events have timestamps expressed in a single timeline they may arrive out-of-order, since the communication channels linking each collector with the monitor introduce different delays. Thus, the monitor needs to ensure that the rule inferences it performs are correct and the out-of-order arrival of events does not cause any false positive or true negative identifications of rule violations.

As an example of this problem consider the following monitoring rule:

```
Rule-2:
∀ t1, t2: Time
Happens(e1, t1, ℜ(t1,t1)) ⇒
Happens(e2, t2, ℜ(t1+1, t1+5))
```

In the case of this rule, it is possible that the monitor receives first an *e2* event and subsequently an *e1* event although the two events were produced with timestamps that would make it possible to unify them with the rule and check if the rule is satisfied (e.g. e1 at T=10 and e2 at T=14). The arrival of *e2* prior of *e1* makes it complex to unify it with `Rule-2` since the rule constraints *e2* to occur with a certain time window after the occurrence of *e1* (i.e. between 1 and 5 time units after the occurrence of e1).

[3] Open source NTP servers can be downloaded from: www.ntp.org.

One possible solution for this problem would be to serialise all events before passing them to the monitor. In this approach

- an Event Serialzer inside the monitoring manager (see Figure 2) stores the maximum channel delay (maxCDi) for each event collector DCi
- If the Event Serializer receives an event Ei from DCi the Event Serializer determines the processing time Tpi (i.e., the time point when the event should be dispatched to the monitor) using the following formula,

  $T^{p}_{i} = T_i + max(maxCD_i) - maxCD_i$, Where T_i is the time stamp of the event E_i.

In the case of the above rule for example, if the event serialiser knows that the maximum transmission delay of the channel that transmits e1 is 4 and the maximum transmission delay of the channel that transmits *e2* is 2, it could determine the processing time of *e2* as 16 as beyond that time point no event from the channel that produces e1 that occurred before *e2* can arrive.

The problem with this solution, however, is that is not safe. This is because it is based on historic estimates of maximum channel delays which may change over time. Thus, there might be cases where the current estimate for the delay of a specific channel is not accurate and, as a result, the serializer may fail to wait long enough for events which have been delayed.

Thus, we have chosen to pass events directly to the monitor as they arrive and store them in the database of past events (see *Database I* in Figure 1). As such, events can be directly used by some rules and if later on an earlier event e' arrives which can be correlated with event *e*, then e is retrieved from *Database I* and the inferences work as if all events had been perfectly serialised.

The problem with this alternative solution is that unless there is a means of identifying the latest time until when an event may be needed when it is stored in the past event database, this database can gradually grow very large and affect the performance of the monitor. For this reason, it is necessary to compute an upper bound of the time until which an event should be stored in the database that we will refer to as the *time-to-live* for the event in the following. In the case of `Rule-2` above, for example, if we assume that there is no other rule to monitor when the event *e2* is received with a timestamp 14 this event should be stored in the event database of the monitor until the time of the channel that produces *e1* events becomes 13.

Thus, in order to handle events which are captured by distributed event collectors and may arrive at the monitor in arbitrary orders, we extended the monitor engine in three additional ways:

- Firstly, we implemented a theory analyser which analyses the set of the formulas which are to be used in monitoring in order to establish upper bounds of the time-to-live for each event. The time-to-live for an event is the time that it should stay in the event data base that is maintained by the monitor in order to ensure that it will be available for consideration for all the formulas that it could be unified with.
- Secondly, we modified the reasoning process of the monitor to ensure that it searches through its event database even for future formulas.
- Thirdly, we implemented a component that scans the event database periodically to remove all the events that have exceeded their upper time-to-live bound.

In the following, we describe the extension of the monitor engine that estimates the upper bound of the time-to-live for each event.

3.2.2.1 Estimation of Upper Time-To-Live Event Bound

The upper time-to-live of an event is the latest time point until which the event should be stored in the database of the monitor to ensure that it will be available for unification with a formula if the need arises.

The estimation of the upper bound of the time-to-live for each event which is fed to a monitoring session is estimated at runtime based on constraints that the formulas which the event can be unified with specify for the time variable of the particular event type and the time variables of the other predicates in the formula.

More specifically, when the monitor receives an event *e* it first tries to unify it with a *Happens* predicate in a formula that has an unconstrained time variable (i.e., a time variable whose upper and lower bounds are not constrained by the values of any other time variables of the predicates in the formula) and expresses the occurrence of an event that has the same type as *e* or a supertype of it. If this succeeds, the monitor creates a new template instance for the formula and searches the event database in order to find other events which may also be unified with the newly created instance. If the event can also be unified with predicates in other formula instances that have constrained time variables the monitor does the unification. Finally, if the event *e* can be unified with predicates which have constrained time variables in other formulas that have not been instantiated yet, *e* is stored in the event database of the monitor. In this case, the event *e* is stored along with a set of upper bounds of its time-to-live (TTL_e) which is defined as

$$TTL_e = \{(\max(TTL_e – clock_1), clock_1), \ldots, (\max(TTL_e – clock_k), clock_k)\} \qquad \text{(F)}$$

where $max(TTL_e–clock_i)$ ($1 \leq i \leq k$) is the maximum time-to-live of the event *e* that is estimated according to clock *i*. A clock *i* in the above formula is the clock of an event collector that produces events which constraint e in different formulas. In the case of `Rule-2` above, for example, a clock would be the clock of the collector that produces *e1* events. Note, however, that the same event collector may be producing different types of events which impose different constraints upon the time variable of an event *e* in different rules. In such cases, the TTL of *e* according to the clock of the specific collector should be computed by taking the maximum of the TTLs for the given clock which are computed by the different rules that include events imposing constraints on *e* and are produced by the collector (hence the use of the operator *max* in (F)).

The upper bound of the time-to-live for an event *e* given the constraints of a specific rule is computed as a solution to a *linear programming problem*. More specifically, assuming that a rule is specified by a formula *f* that has *n* time variables, the lower and upper bounds of each of the time variables T_i of it are defined by linear inequalities of the following forms:

(LB): $a_{Li1}T_1 + a_{Li2}T_2 + \ldots + a_{Lin}T_n + b_{Li} \leq T_i$

(UB): $T_i \leq a_{Ui1}T_1 + a_{Ui2}T_2 + \ldots + a_{Uin}T_n + b_{Ui}$

A_f						
-1	a_{L12}				$a_{L1(n-1)}$	a_{L1n}
a_{L21}	-1				$a_{L2(n-1)}$	a_{L2n}

B_f
$-b_{L1}$
$-b_{L2}$

$a_{L(n-1)1}$	$a_{L(n-1)2}$				-1	$a_{L(n-1)n}$		$-b_{L(n-1)}$
a_{Ln1}	a_{Ln2}				$a_{Ln(n-1)}$	-1		$-b_{Ln}$
1	$-a_{U12}$				$-a_{U1(n-1)}$	$-a_{U1n}$		b_{U1}
$-a_{U21}$	1				$-a_{U2(n-1)}$	$-a_{U2n}$		b_{U2}
$-a_{U(n-1)1}$	$-a_{U(n-1)2}$				1	$-a_{U(n-1)n}$		$b_{U(n-1)}$
$-a_{Un1}$	$-a_{Un2}$				$-a_{Un(n-1)}$	1		b_{Un}

Figure 3 – Time Constraint Specification Matrices

The form of these constraints is imposed by EC-Assertion which requires that a range [LB,…,UB] is defined for each of the time variables of the *Happens* predicates in a formula (see [1]) and LB, UB are defined by linear functions over the values of one or more other time variables in the formula and/or constants. The above inequalities (LB) and (UB), however, can be re-written as:

(LB'): $a_{Li1}T_1 + a_{Li2}T_2 + \ldots + a_{Li(i-1)}T_{(i-1)} + (-1)T_i + a_{Li(i+1)}T_{(i+1)} + \ldots + a_{Lin}T_n \leq -b_{Li}$

(UB'): $-a_{Ui1}T_1 - a_{Ui2}T_2 - \ldots - a_{Ui(i-1)}T_{(i-1)} + T_i - a_{Ui(i+1)}T_{(i+1)} - \ldots - a_{Uin}T_n \leq b_{Ui}$

When *e* is received, the value of the time variable of the predicate that *e* can be unified with can be further restricted to be equal to the timestamp of *e*. Following this, it is possible to compute the maximum time for each of the events that could be unified with the other predicates of the formula. This is possible by finding the maximum possible value for each the remaining time variables T_j of the formula given the constraints $A_f\,T_f \leq B_f$ and $T_f \geq 0$ where T_f is the vector of the time variables of *f* and A_f and B_f are the matrices shown in Figure 3 which are constructed based on (LB') and (UB'):

The computation of the maximum possible value for each the remaining time variables T_j of the formula is equivalent to finding a solution to the linear programming problem

$$\textit{maximise } T_j \textit{ subject to } A_f\,T_f \leq B_f \textit{ and } T_f \geq 0$$

A solution to the above problem can be found by using the classic *Simplex algorithm* or one of its variants [3].

The matrices A_f and B_f are constructed for each formula prior to the start of the monitoring session according to the algorithm *ConstructLinearInequalitiesMatrix(f, A_f, B_f)* that is defined below. This algorithm constructs A_f and B_f based on the forms (LB') and (UB') of the constraints of the time variables of a formula *f*.

```
ConstructLinearInequalitiesMatrix(f, $A_f$, $B_f$)
BEGIN
Let $TM_f$ = {$T_1$,…,$T_n$} be the set of the time variables in f;
Let $A_f$ be an 2n × n matrix and $B_f$ be an n × 1 matrix representing the boundary constraints of the variables in TM in the standard form of a linear programming problem
FOR each time variable $T_i$ in $TM_f$ DO
  IF the lower boundary of $T_i$ is defined by the formula
    "$a_{Li1}T_1 + a_{Li2}T_2 + … + a_{Lin}T_n + b_{Li} \leq T_i$" THEN
    FOR k=1 to i-1 DO
      $A_f(i,k) = a_{Lik}$
    ENDFOR
    $A_f(i,i) = -1$
    FOR k=i+1 to n DO
      $A_f(i,k) = a_{Lik}$
    ENDFOR
    $B_f(i,1) = -b_{Li}$
  END IF
  IF the upper boundary of $T_i$ is defined by the formula
    "$T_i \leq a_{Ui1}T_1 + a_{Ui2}T_2 + … + a_{Uin}T_n + b_{Ui}$" THEN
    FOR k=1 to i-1 DO
      $A_f (i,k) = -a_{Uik}$
    ENDFOR
    $LB_f (i,i) = 1$
    FOR k=i+1 to n DO
      $A_f (i,k) = -a_{Uik}$
    ENDFOR
    $B_f(i+k,1) = b_{Ui}$
  END IF
END
```

Following the construction of the matrices A_f and B_f prior to the start of a monitoring session, at runtime the upper boundary of TTL_e is computed according to the algorithm *ComputeTTL(E, TTL_E)* below.

```
ComputeTTL(E, TTL_E)
BEGIN
   TTL_E = {}
   FOR EACH formula f that has a predicate P with a constrained time variable
   which can be unified with E DO
      Let k be the number of time variables of f
      Let T_E be the time variable of P
      Let A_f and B_f be the time constraint matrix of f and I be the index of T_E
      in them
      RewriteAB(A_f , B_f, timestamp(E), I, k, newA_f , newB_f)
      flag = 1
      WHILE there are more time variables T in F other than T_E AND (flag = 1) DO
         maximise(T, newA_f, newB_f, maxT, flag)
         /* flag: is a variable taking the values 1 or 0 to indicate
            if:
            - a solution has been found by the routine maximise(1),
            - no solution exists (0)
         */
         IF flag = 1 THEN
            IF exists (maxE,c) in TTL_E such that c == clock(T) THEN
               IF maxT > maxE THEN
                  maxE = maxT
               ENDIF
            ELSE
               TTL_E = TTL_E ∪ {(maxT, clock(T))}
            ENDIF
         ENDIF
      ENDWHILE
   END FOR
END
```

```
RewriteAB(A_f, B_f, timestamp(E), I, k, newA_f, newB_f)
BEGIN
   newA_f = A_f
   newB_f = B_f
   FOR j=1 to k DO
      IF newA_f[j,I] ≠ 0 THEN
         newB_f[j] = newB_f[j] – newA_f[j,I]*timestamp(E)
```

```
            newA_f[j,I] = 0
        ENDIF
    ENDFOR
END
```

The algorithm *ComputeTTL(E, TTL$_E$)* first rewrites the matrices A_f and B_f in order to transform the time constraints based on the known value of the time variable of *P* in *f* (this value is the timestamp of the event *E* that can be unified with *P*). The transformation is done by the routine *RewriteAB*. For all the constraints where the time variable of *P* that can be unified with the current event *E* appears with a non 0 coefficient in the matrix A_f, *RewriteAB* adds the product of the coefficient and the timestamp of *E* to the corresponding value in matrix B_f and sets the relevant coefficient in A_f to 0. Thus, if, for example, the respective time variable is T_i, *RewriteAB* will rewrite the constraints of the form:

(LB'): $a_{Li1}T_1 + a_{Li2}T_2 + \ldots + a_{Li(i-1)}T_{(i-1)} + a_{Lii}T_i + a_{Li(i+1)}T_{(i+1)} + \ldots + a_{Lin}T_n \leq -b_{Li}$

(UB'): $-a_{Ui1}T_1 - a_{Ui2}T_2 - \ldots - a_{Ui(i-1)}T_{(i-1)} - a_{Uii}T_i - a_{Ui(i+1)}T_{(i+1)} - \ldots - a_{Uin}T_n \leq b_{Ui}$

as

(LB''): $a_{Li1}T_1 + a_{Li2}T_2 + \ldots + a_{Li(i-1)}T_{(i-1)} + 0T_i + a_{Li(i+1)}T_{(i+1)} + \ldots + a_{Lin}T_n \leq -b_{Li} - a_{Lii}\,\text{timestamp}(E)$

(UB''): $-a_{Ui1}T_1 - a_{Ui2}T_2 - \ldots - a_{Ui(i-1)}T_{(i-1)} - 0T_i - a_{Ui(i+1)}T_{(i+1)} - \ldots - a_{Uin}T_n \leq b_{Ui} + a_{Uii}\,\text{timestamp}(E)$

Following this transformation, *ComputeTTL(E, TTL$_E$)* finds the maximum possible values of the time variables for the remaining predicates in the formula and sets the corresponding elements of the set of TTL_e to the maximum of these values. The found values represent the maximum time at which events unifiable with the predicate that is associated with the respective time variable can occur.

In the following, we explain how TTL is computed by the above algorithm using two examples. Our first example is about a monitoring session in which the following rule is to be monitored:

```
Rule-3:
∀ t1, t2, t3: Time
Happens(e1, t1, ℜ(t1,t1)) ∧
Happens(e2, t2, ℜ(t1+1, t2))⇒
Happens(e3, t3, ℜ(t2+1, t3))
```

The constraints for the time variables of `Rule-3` are as follows:

(C1): $t1 + 1 \leq t2$

(C2): $t2 + 1 \leq t3$

These constraints are expressed in the form assumed by the Simplex method as follows:

(C1'): $1t1 - 1t2 + 0t3 \leq -1$

(C2'): $0t1 + 1t2 - 1t3 \leq -1$

Assuming that an event e3 is received at time t3 = 10, (C1') and (C2') are transformed into:

(C1''): $1t1 - 1t2 + 0t3 \leq -1$

(C2''): $0t1 + 1t2 - 0t3 \leq -1 + 10$

and the maximum value for t1 is 8 and for t2 is 9.

Thus, the TTL of e3 would be {(8, e1), (9, e2)}[4]. Note that, in this example, if *e3* had been received at t3=1 there would be no solution to the problem and, therefore, the event *e3* in this case would not have to be maintained for `Rule-3`. It should be appreciated, however, that in this case *e3* could still be recorded in the event database of the monitor if there was another formula for which there was a solution.

Assume now another monitoring session where in addition to `Rule-3`, `Rule-4` below should also be monitored.

```
Rule-4:
∀  t1, t2, t3: Time
Happens(e1, t1, ℜ(t1,t1)) ∧
Happens(e4, t2, ℜ(t1+5, t1+7))⇒
Happens(e3, t3, ℜ(t2+2, t3))
```

The constraints for the time variables of `Rule-4` are:

(C3): $t1 + 5 \leq t2$

(C4): $t2 \leq t1 + 7$

(C5): $t2 + 2 \leq t3$

or equivalently in the form assumed by Simplex :

(C3'): $1t1 - 1t2 + 0t3 \leq -5$

(C4'): $-1t1 + 1t2 + 0t3 \leq 7$

(C5'): $0 t1 + 1t2 - 1t3 \leq -2$

Thus, assuming that an event *e3* is received at time t3 = 10, (C3') − (C5') are transformed into the following constraints by *RewriteAB*:

[4] In the examples of this section, we use the event types *(e1, e2, …)* to refer to the event collectors that produce the respective events and the clocks of these collectors. This is necessary as the specifications of events in the rules used in our examples do not refer to the collectors which capture the events.

(C3''): $1t1 - 1t2 + 0t3 \leq -5$

(C4''): $-1t1 + 1t2 + 0t3 \leq 7$

(C5''): $0\,t1 + 1t2 - 0t3 \leq 8$

Based on (C3'') – (C5''), the TTL of *e3* due to t1 (i.e., the time variable or clock of e1) would be 8 and the TTL of *e3* due to t2 (i.e., the time variable or clock of *e4*) would be 9. Thus, in cases where both `Rule-3` and `Rule-4` are being monitored the algorithm *ComputeTTL*, would set the TTL of *e3* to {(8, e1), (9, e2), (9, e4)}.

Note that the algorithm *ComputeTTL(E, TTL_E)* considers all event types in formulas that can be unified an event *e*. In this process, an event type is considered as the vector of types of the event's arguments and its result. Thus, an event can be unified with events in formulas whose type is a supertype of its own or, equivalently, all event types which can be produced by considering the supertypes of one or more of the event's arguments. For example, if we have an event that represents the call of the operation *log(Car v, Employee p): Boolean*, then its type is *"log" × Car × Employee × Boolean*. The type of this event is a subtype of the event types *"log" × Vehicle × Person × Boolean*, *"log" × Vehicle × Employee × Boolean*, and *"log" × Car × Employee × Boolean*, assuming that *Vehicle* is a supertype of *Car* and *Person* is a supertype of *Employee*. Consequently, formulas with the latter event types will also be considered.

Following the estimation of TTL_e for an event *e*, *e* is stored in the event database of the monitor. Subsequently, when the time-to-live for an event *e* expires, that is when the latest recorded times of the clocks (channels) that constraint the event exceed its TTL_e, the event is deleted from the database. The guarantee that an event with a timestamp that is less than the timestamp of the latest event which has been received from a channel cannot occur is due to the fact that we assume event channels operating using the TCP/IP protocol and, therefore, the events received from these channels arrive at the same order as the order in which they are dispatched.

3.2.3. *Optimization of the Monitoring Process by Template Pruning*

The main area that we looked at was how to reduce the increase in the number of active templates during the monitoring process. This investigation focused at the process of template creation and the possibility of pruning active templates which do not provide sufficient information for making a decision about the rule instance that they represent and cannot be possibly updated by further events. In the monitoring scheme implemented by the monitor, a new instance of a template is created if the variable bindings of a predicate have values that are different from the event variable values during the unification. This process may create many partially instantiated template instances which are not needed in the monitoring process.

As an example of such cases, consider `Rule-5` below:

```
Rule-5

∀ t1 :Time, ∃  t2 :Time

Happens(ev(_eID,    _sender1,    _receiver1,    REQ-B,    isAvailable(    _car,
_loc),_sender1),t1,R(t1,t1)) ^

Happens(ev(_eID,    _receiver1,    _sender1,    RES-A,    isAvailable(    _car,
_loc),_receiver1),t2, R(t1,t2))

⇒ oc:self:sub(t2,t1) < 500
```

When an event that represents the invocation of the operation *isAvailable*, with unique ID say *isAvail120*, is encountered the monitor creates a new template instance and unifies the event with the predicate `Happens(ev(_eID, _sender1, _receiver1, REQ-B, isAvailable( _car, _loc),_sender1),t1,R(t1,t1))` in the rule. Subsequently, when an event that represents the response from the particular invocation of the operation *isAvailable* is encountered, the monitor unifies it with the predicate `Happens(ev(_eID, _receiver1, _sender1, RES-A, isAvailable( _car, _loc),_receiver1),t2, R(t1,t2))` in the template instance of the rule that it created when the invocation event occurred. However, to cover the possibility of having another response to the same operation call with different variable values, v1 of DVP also creates a another copy of the template that represents the partially instantiated rule following the invocation. Although this functionality is in general necessary in order to ensure the completeness of the reasoning process implemented by the monitor, the creation of the template copy in this particular example (and, in fact, all cases which refer to the atomic invocation and response of a synchronous operation) is not necessary. This is due to the semantics of the operation invocation in the particular example. More specifically, in the case of `Rule-5` there can be only one response to the call of the operation `isAvailable`. To address this issue we amended the template creation process in the case of unification of templates with events that represent responses from operation invocations. This optimisation has led to dramatic improvement in the monitoring results which was discussed in [5].

3.3. Other Limitations

In A4.D3.1 [1], we had summarised some limitations of the first version of the Dynamic Validation Prototype that could be addressed in the second version subject to further assessment and prioritisation. In this section, we revisit those limitations and give an overview of whether they were eventually addressed along with a rationale for the way that we treated them.

3.3.1. *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, `Happens(e, t1) ⇒ HoldsAt(f, t2) ∧ t1 ≤ 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.

3.3.2. *Lack of Multiple Inheritance*

Limitation: 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.

Action: As discussed in A4.D3.1 [1], 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.

3.3.3. Support for Logical Clocks

Limitation: As explained earlier, DVPv1 assumed the all the events that were fed to the monitor had time stamps generated by the same clock and arrive from the event collectors that captured them to the monitor in the very order that is imposed by their timestamps. These assumptions do not hold in a distributed system and the possibility of using a vector of event collector and monitor clocks, similarly to the concept of logical clocks had been referred to as a potential solution to the problem in [1].

Action: The implementation of clock synchronisation and the amendments that we introduced to the monitor in order to deal with possible delays in the transmission of events have addressed this problem without the need to rely on logical clocks.

3.3.4. Support for Non-Determinism

Limitation: DVPv1 has no support for non-determinism since all the fluents which are generated during the monitoring process have a specific value always. In [1], we had identified the possibility of offering support for non-determinism by supporting the use of the predicate of EC `Releases` [8] as a means of reasoning about systems whose internal behaviour is not known.

Action: We have decided not to support this predicate as in monitoring typical security and dependability properties we have not identified a need to support ambiguities in fluent values of the above kind.

3.3.5. Support for a Full Object-Oriented Database

Limitation: The databases used in DVPv1 for storing past-time events and Initiates/Terminates predicates were simple, in-memory databases that could by a full-fledged Object-Oriented database, for example *db4objects*[5], to enable a more efficient monitoring process.

Action: Our initial experimental and evaluation results have not indicated severe performance problems that would require the implementation of this feature. Hence, although it might be potentially useful, this feature was not implemented.

3.3.6. Minimisation of Past-Time Predicates in the DB

Limitation: 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.

[5] http://www.db4objects.com/community/

Action: This feature has been implemented through the computation of the time-to-live for events and the deletion of events which can no longer be used in the reasoning process as explained earlier.

3.3.7. Calling External Functions

Limitation: In order to be able to support control and recovery, DVP 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.

Action: This feature was not implemented as part of DVPv2 as it relates to control and recovery. An implementation of it will be provided along with the recovery mechanisms which are due on month 32 (A4.D6.1).

3.3.8. Supporting Internal Alarms

Limitation: 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.

Action: As it relates to control and recovery this feature will be reconsider in the context of producing the deliverable A4.D6.1.

3.4. Future Work

Future work related to the Dynamic Validation Prototype will focus on the integration of:

- the threat detection mechanisms
- the mechanisms for providing diagnosis for violations of S&D property monitoring rules

The above mechanisms are currently under development in the project and initial versions of them are due in month 24. Following the release of their initial implementations, these mechanisms will be integrated into DVP to provide a comprehensive monitoring platform.

4. Installation and Usage Guide

4.1. Required Software

To use the dynamic validation prototype, 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.

4.2. 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).

4.3. How to Use the Prototype

4.3.1. 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 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`

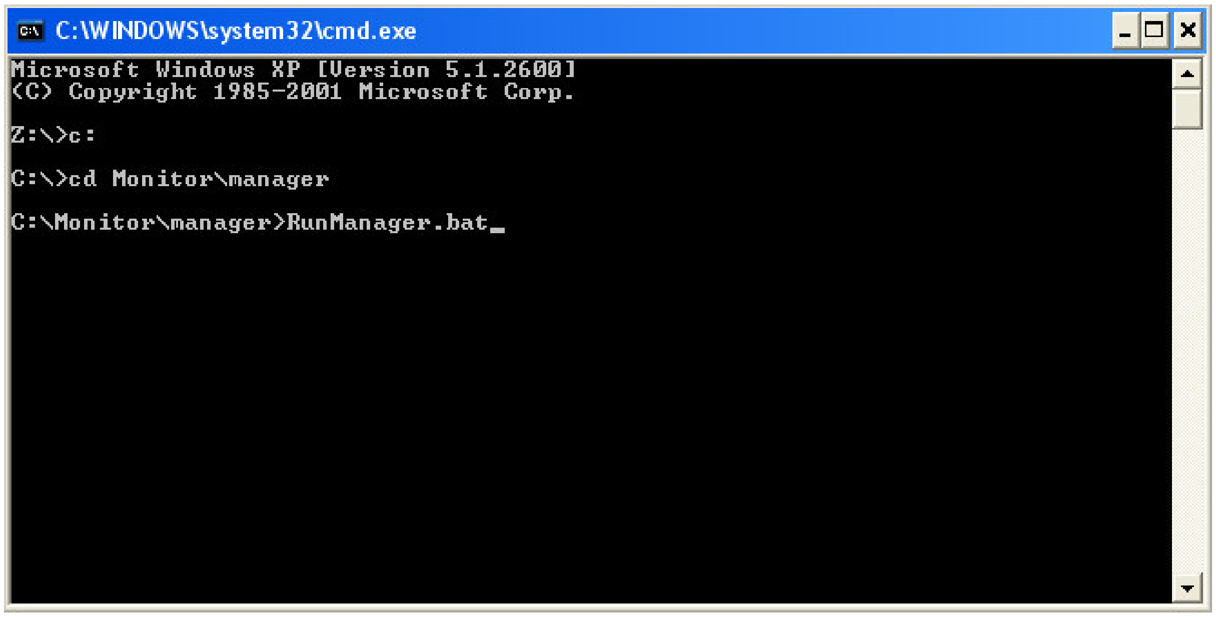
Section 4.3.2 describes how to use City monitoring manager and event collector.

4.3.2. 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 event 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 Figure 4 below. Following this, the monitor manager window will pop up.



```
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 4 – Command prompt window

2. Then, to import the formulae to be monitored, select the option "Import Formulae" from the "File" menu of the manager.
3. In the file opening dialog box that appears following the selection of this option (see Figure 5), choose the XML file that contains the formulae that you want to monitor.

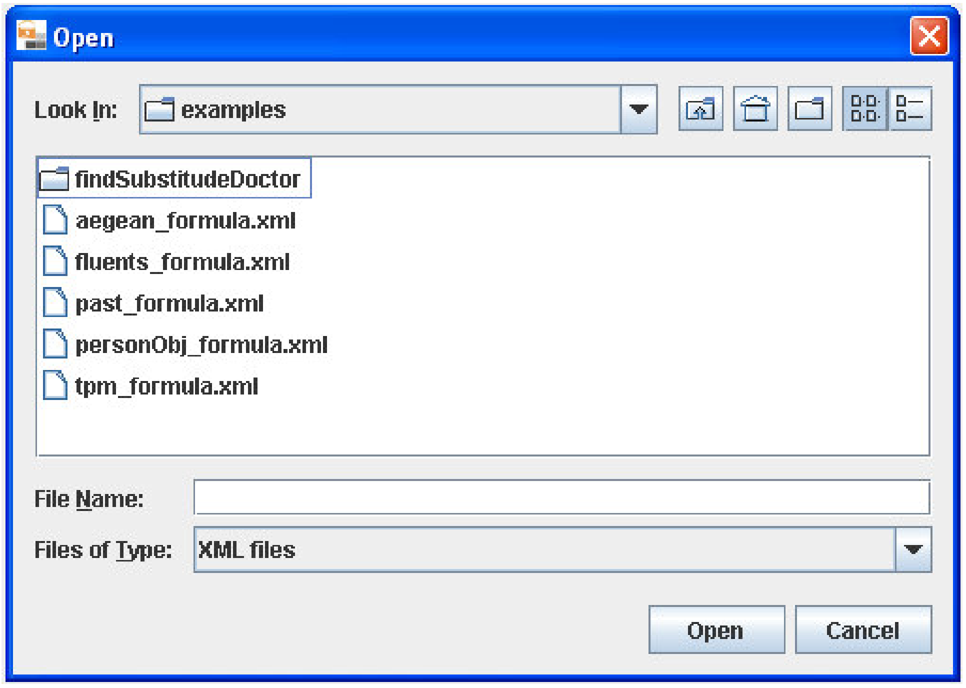
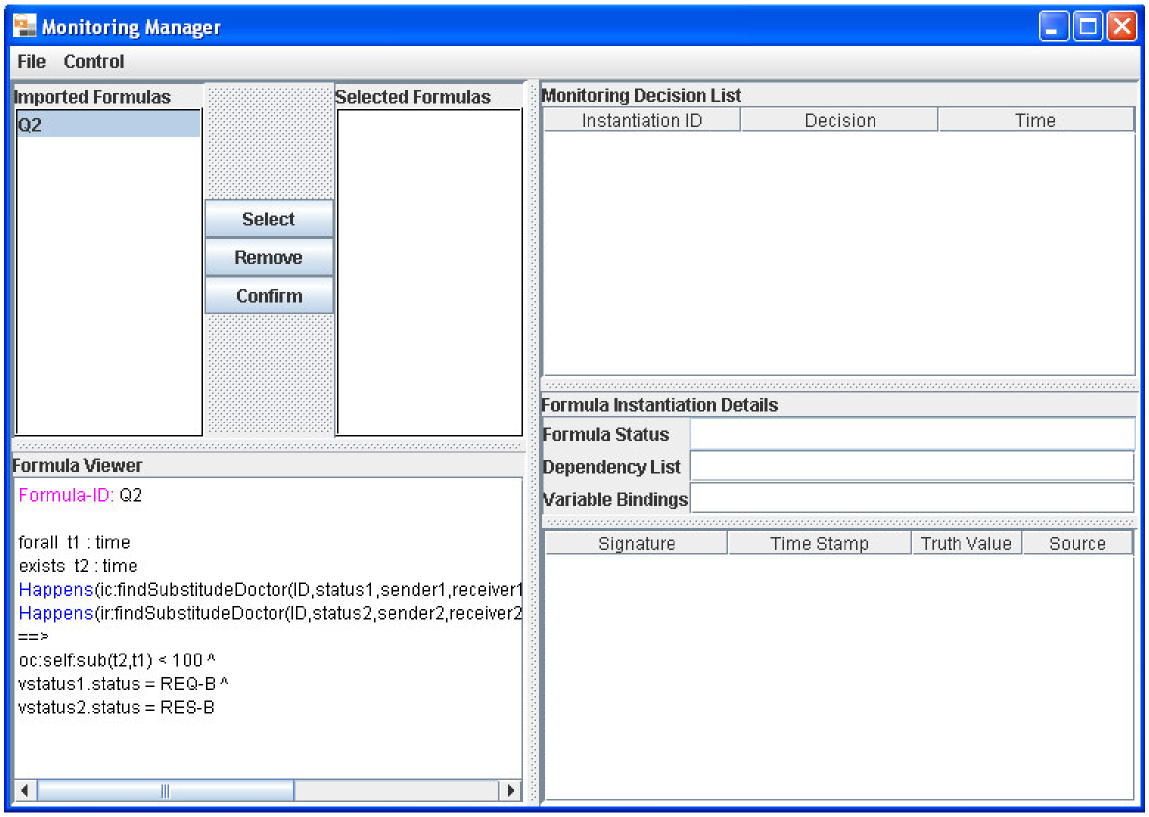


Figure 5 – The file opening dialogue box

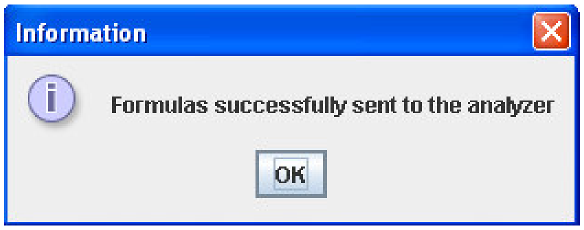
4. The monitor manager will then read all the formulae from the file and display the formulae in the display panel of the monitoring manager as illustrated in Figure 6.



```
Formula-ID: Q2

forall t1 : time
exists t2 : time
Happens(ic:findSubstitudeDoctor(ID,status1,sender1,receiver1
Happens(ir:findSubstitudeDoctor(ID,status2,sender2,receiver2
==>
oc:self:sub(t2,t1) < 100 ^
vstatus1.status = REQ-B ^
vstatus2.status = RES-B
```

Figure 6 – The Monitoring Manager

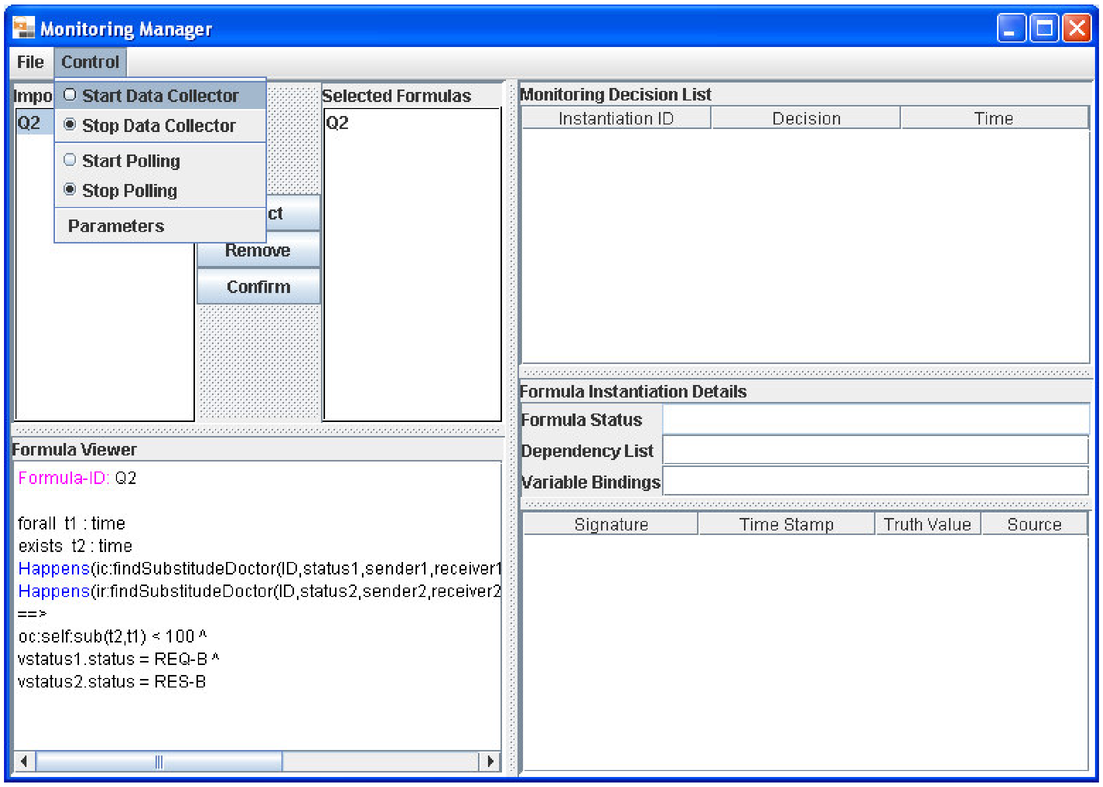
The monitoring manager lists the identifiers of the imported formulae in the "Imported Formulae" panel. To view a formula in the event calculus format, the user may select its ID. Following this selection, the formula with the selected ID will be shown in the "Formula Viewer" panel of the manager. If 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 (see Figure 7).



Formulas successfully sent to the analyzer

Figure 7 – Formulas submission message

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 [2]. 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; commons-net-1.4.1.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;commons-net-1.4.1.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 option from the Control menu (see Figure 8). Any attempt to invoke a web service in the port 8081 will result to report this event to the monitor manager tool.

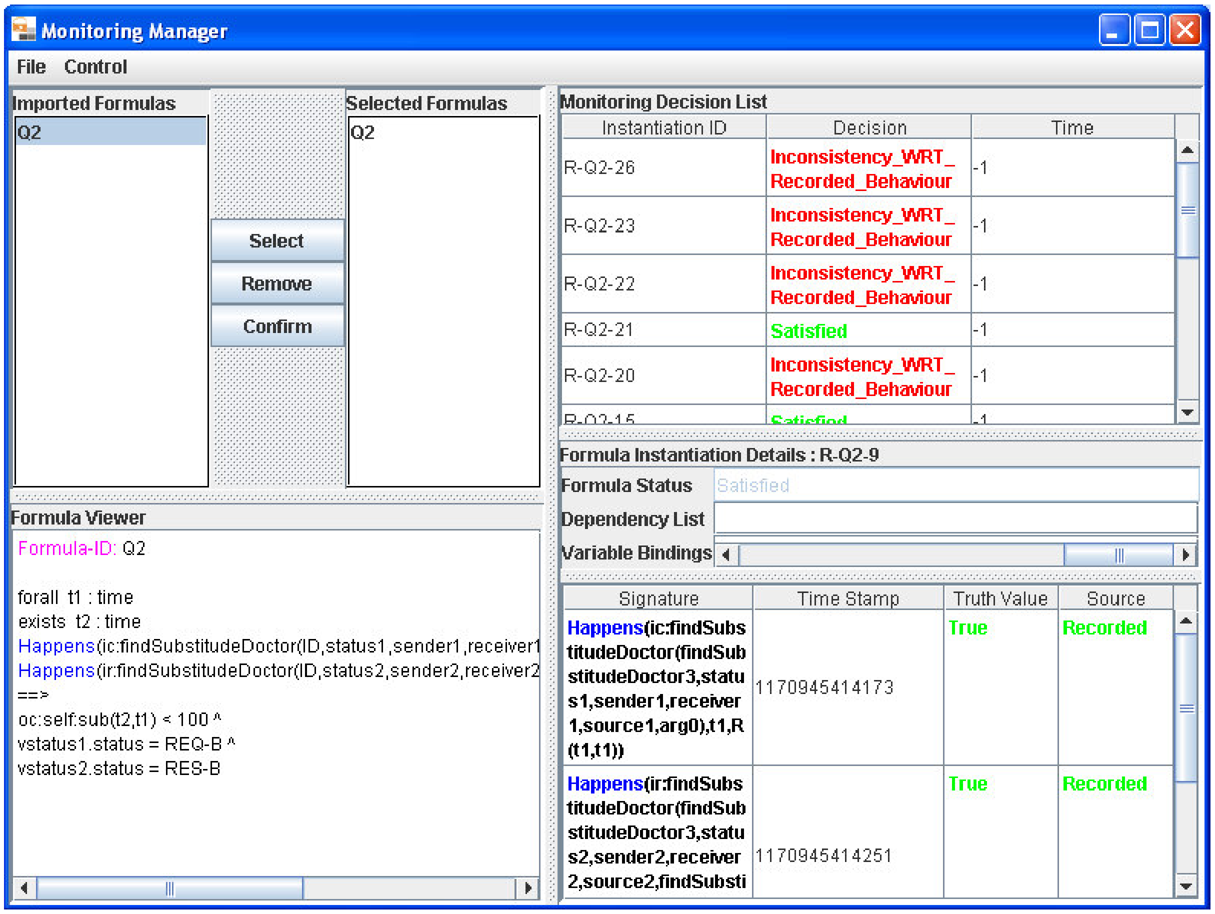


```
Formula-ID: Q2

forall t1 : time
exists t2 : time
Happens(ic:findSubstituteDoctor(ID,status1,sender1,receiver1
Happens(ir:findSubstituteDoctor(ID,status2,sender2,receiver2
==>
oc:self:sub(t2,t1) < 100 ^
vstatus1.status = REQ-B ^
vstatus2.status = RES-B
```

Figure 8 – Starting and stopping 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 (see Figure 8)
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 9. 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.



Monitoring Decision List

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

Formula Instantiation Details : R-Q2-9

Formula Status: Satisfied

Dependency List:

Variable Bindings:

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

Formula Viewer

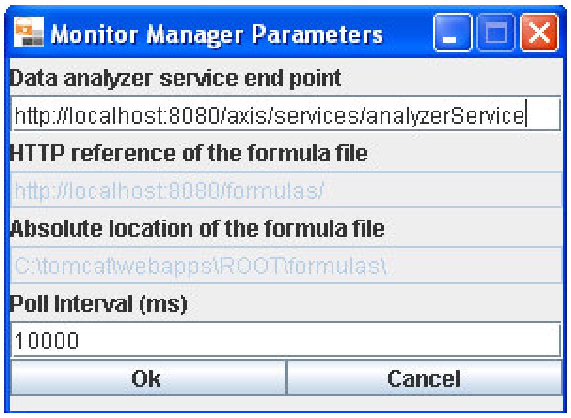
```
Formula-ID: Q2

forall t1 : time
exists t2 : time
Happens(ic:findSubstituteDoctor(ID,status1,sender1,receiver1
Happens(ir:findSubstituteDoctor(ID,status2,sender2,receiver2
==>
oc:self:sub(t2,t1) < 100 ^
vstatus1.status = REQ-B ^
vstatus2.status = RES-B
```

Figure 9 – Monitoring Decision and Formula Instantiation Panels

8. 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 (see Figure 9). This panel displays the formula status, other formulae that the specific formula may depend

on (see [8] 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 [8]).

9. 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 (see Figure 10), specify a new value for the polling interval. Subsequently, the user should select the option "restart polling" from the control window.



Data analyzer service end point
http://localhost:8080/axis/services/analyzerService
HTTP reference of the formula file
http://localhost:8080/formulas/
Absolute location of the formula file
C:\tomcat\webapps\ROOT\formulas\
Poll Interval (ms)
10000

Figure 10 – Monitor Manager parameters dialogue box

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

4.4. Example of Using the DVP Together With multiple Event Collectors

In *DVPv2* data collectors are identified by the IP address of the machine hosting the data collector. Therefore it is recommended that each data collector should be deployed on a different machine. In section 4.3 we discussed the command line arguments needed to start the data collector. We review the arguments once again here,

```
java -cp xercesImpl.jar;commons-net-1.4.1.jar; code.TcpTunnel
listeningPort webserviceHost webservicePort monitorManagerHost
monitorManagerHost
```

listeningPort : This is the port the data collector is listening to. The default value of this port is 8081

webserviceHost: This is the IP address of the machine that hosts the actual web service. We recommend to use the actual IP address of a machine as the value of this parameter, rather than *localhost* or *127.0.0.1*

webservicePort: This is the port number that the web service container listening to.

monitorManagerHost: This is the IP address of the machine hosting the monitor manager. We recommend to use the actual IP address of a machine as the value of this parameter, rather than *localhost* or *127.0.0.1*

monitorManagerPort: This is the port number that the monitor manager is listening to. The default value of this port is 12345.

In the rest of 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 4.2 .
- In the same way as in the previous step deploy the service located in C:\Monitor\examples\findSubstitudeDoctor, 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 Polling options from the Controls menu of the monitor manager.
- Execute the RunTestDRI.bat file from the C:\Monitor\examples\findSubstitudeDoctor folder which invokes the DoctorRegistryInquiryPT web service.

5. Conclusion

This report has provided an overview of the first version of the Dynamic Validation Prototype (DVP) which has been developed by City University in SERENITY (as part of the deliverable A4.D3.1) and then describes the amendments that were introduced to this prototype as part of its second version. The latter version along with this report constitutes the deliverable A4.D3.3 of SERENITY.

This report has focused on the extensions that were implemented in the second version of DVP. These included:

— Changes in the functionality of the various Event Collectors and the Monitor engine to enable the synchronisation of the clocks of the former with the clock of the latter component when Event Collectors are distributed on different machines.

— New functionality in the Monitor Engine, to deal with the problem of unordered receipt of events by the monitor that might be caused by delays in the transmission of events from distributed event collectors.

— Optimisations of the monitoring process that enable the pruning of active templates in order to reduce the average time of making a decision about the satisfaction of a formula.

The document also revisits limitations that had been identified for the first version of DVP, and indicates whether action has been taken to address them. Finally, this report contains a guide for the installation and usage of the second version of DVP that has been delivered as part of A4.D3.3.

References

[1] Androutsopoulos K., Ballas K., Kloukinas C., Mahbub K., and Spanoudakis G. (2006): V1 of Dynamic Validation Prototype, Deliverable A4.D3.1, SERENITY Project

[2] Campadello S., Compagna L., Gidoin D., Giorgini P., Holtmanns S., Latanicki J., Meduri V., Pazzaglia J.-C., Seguran M., Thomas R., and Zanone N. (2006): S&D Requirements specification, Deliverable A7.D2.1, SERENITY Project

[3] Cormen T., Leiserson C., Rivest R., and Stein S. (2001): Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill. ISBN 0-262-03293-7.

[4] 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

[5] Kloukinas C., Mahbub K., and Spanoudakis G. (2007): Evaluation of V1 of Dynamic Validation Prototype, Deliverable A4.D3.2, SERENITY Project

[6] Maña A, Muñoz A, Sánchez F, Serrano D. (2006): Patterns and Integration Schemes Languages, Deliverable A5.D2.1, SERENITY Project

[7] NTP, www.ntp.org (last seen on 3/8/2007)

[8] Shanahan, M. P. (1999): The Event Calculus Explained, in Artificial Intelligence Today, LNAI no. 1600:409-430, Springer

[9] Spanoudakis G., Kloukinas C., Androutsopoulos K. (2006): Towards Security Monitoring Patterns, 22nd Annual ACM Symposium on Applied Computing, Technical Track on Software Verification (to appear)