Chapter 13
The Runtime Monitoring Framework of SERENITY

George Spanoudakis, Christos Kloukinas and Khaled Mahbub

Abstract This chapter describes SERENITY's approach to runtime monitoring and the framework that has been developed to support it. Runtime monitoring is required in SERENITY in order to check for violations of security and dependability properties which are necessary for the correct operation of the security and dependability solutions that are available from the SERENITY framework. This chapter discusses how such properties are specified and monitored. The chapter focuses on the activation and execution of monitoring activities using S&D Patterns and the actions that may be undertaken following the detection of property violations. The approach is demonstrated in reference to one of the industrial case studies of the SERENITY project.

George Spanoudakis
Dept. of Computing, City University,
Northampton Square, London, EC1V 0HB, e-mail: G.Spanoudakis@soi.city.ac.uk

Christos Kloukinas
Dept. of Computing, City University,
Northampton Square, London, EC1V 0HB, e-mail: C.Kloukinas@soi.city.ac.uk

Khaled Mahbub
Dept. of Computing, City University,
Northampton Square, London, EC1V 0HB, e-mail: K.Mahbub@city.ac.uk

13.1. Introduction

Ensuring the security and dependability of complex systems operating in highly distributed environments and frequently changing contexts, whilst maintaining system interoperability and adaptability, is one of the major challenges of current research in the area of security and dependability [22]. This is because, as operational conditions change, the security and dependability mechanisms of a system may become ineffective and, when this happens, the system will need to adapt or replace them to ensure the preservation of the desired security and dependability (S&D) properties. In such circumstances, the ability to react dynamically requires the monitoring of the operation of the security and dependability mechanisms that are deployed by the system and the identification of conditions indicating the compromise of security and dependability properties. These needs are prominent especially in systems with distributed components that are deployed over changing infrastructures and communicate over heterogeneous and changing networks.

This chapter presents the monitoring framework that has been developed in SERENITY to enable the monitoring the security and dependability mechanisms at runtime.

As discussed in previous chapters of this book, one of the key objectives of SERENITY has been the development of a runtime framework, known as *SERENITY Runtime Framework (SRF)*, enabling systems which operate in dynamic environments to configure, deploy and adapt mechanisms for realising S&D Properties dynamically. In particular, the SRF supports the dynamic selection, configuration and deployment of components that realise S&D Properties according to *S&D Patterns*. An S&D Pattern in SERENITY specifies a reusable S&D Solution for realising a set of S&D Properties. It also specifies the contextual conditions under which this solution becomes applicable, and invariant conditions that need to be monitored at run-time in order to ensure that the solution described by the pattern behaves correctly. A set of S&D Patterns describing the same application interface and offering the same S&D Properties forms an S&D Class.

When an application needs to deploy a solution that realises specific S&D Properties using a specific API, it asks the SRF for patterns that can provide the required properties and belong to an S&D Class compatible with the required API. The SRF searches through its library of S&D Patterns and, if such patterns exist that are applicable in the current context of the application, selects one of them and returns a reference to its implementation to the application. Subsequently, the application uses the selected implementation through calls to the API it had requested.

During the deployment of an S&D Pattern by an application, it is necessary to monitor whether the invariant conditions specified in the pattern are satisfied and take corrective actions if a violation of these conditions is identified. The monitoring of these conditions is the responsibility of the monitoring framework that is discussed in this chapter.

The monitoring framework of SERENITY is called EVEREST (*EVE*nt *RE*a-*S*oning *T*oolkit). EVEREST is available as a service to the SRF and when an S&D Pattern is activated it undertakes responsibility for checking conditions regarding the runtime operation of the components that implement the pattern. These conditions are specified within S&D Patterns by *monitoring rules* expressed in *EC-Assertion*, i.e., a temporal formal language based on Event Calculus [27]. EVEREST can detect violations of monitoring rules against streams of runtime events, which are sent to it by different and distributed event sources, the *Event Capturers*. It also has the capability to: *(i)* deduce information about the state of the system being monitored, by using *assumptions* about the behaviour of a system and how runtime events may affect its state, *(ii)* detect potential violations of monitoring rules (known as *threats*), by estimating belief measures in the potential of occurrence of such violations, and *(iii)* perform diagnostic analysis in order to identify whether the events causing a violation are genuine or the result of a system fault or an attack. This chapter focuses on the basic monitoring capabilities of EVEREST and the support that it provides for reacting to violations of monitoring rules. The threat detection and diagnostic capabilities of EVEREST are beyond the scope of this chapter and are discussed in the next chapter of this book [33].

The rest of this chapter is structured as follows. Section 13.2 introduces a scenario demonstrating the need for runtime monitoring of the security and dependability mechanisms of a system at runtime. Section 13.3 provides an overview of the architecture of EVEREST and its relation with the SERENITY runtime framework. Section 13.4 presents the language for specifying monitoring rules as part of S&D Patterns. Section 13.5 discusses the core monitoring capabilities of EVEREST. Section 13.6 provides an overview of the implementation of EVEREST and results of experimental evaluations that have been conducted to evaluate it. Finally, Section 13.7 reviews related work and, Section 13.8 concludes by identifying aspects of EVEREST that require further research and development.

13.2. A Scenario for Runtime Monitoring of Security and Dependability

To appreciate the need for monitoring and adaptation of system security and dependability mechanisms at runtime, consider a system which manages access to different resources of an organisation, through a combination of user authentication, device identification and location detection capabilities [3].

In this system, referred to *Location Based Access Control System* (*LBACS*) in the following, users entering and moving within the premises of an organisation using mobile computing devices (e.g., a notebook or smart phone) may be given access to different resources, such as the enterprise intranet, printers or the Internet, depending on their user-id, the id of the mobile device that they are using, and the location of this device. Resource access is granted depending on policies, which determine when access to a particular type of resource is considered to be harmful or not. A policy may, for example, determine that an authenticated employee of the organisation who is trying to access a printer via the local wireless network, whilst being in an area of the premises that is accessible to the public, should be granted access, whilst authenticated visitors should only be given access to printers when they are in one of the organisation's meeting rooms.

The general architecture of LBACS is shown in Fig. 13.1. As shown in the figure, the access control solution of LBACS is based on two servers: a *location* and a *control* server. The control server polls the location server at regular intervals, in order to obtain the position of the devices of all the users who are currently connected to the system. The location server calculates the position of different user devices from signals that it receives from devices through location sensors. The estimates of device positions are not exact and are associated with an accuracy measure. The authentication of the identity of the different user devices is based on the existence of a TPM chip on them and its ability to respond to requests by the authentication server of the system (LBACS is discussed in more detail in Chapter 18).

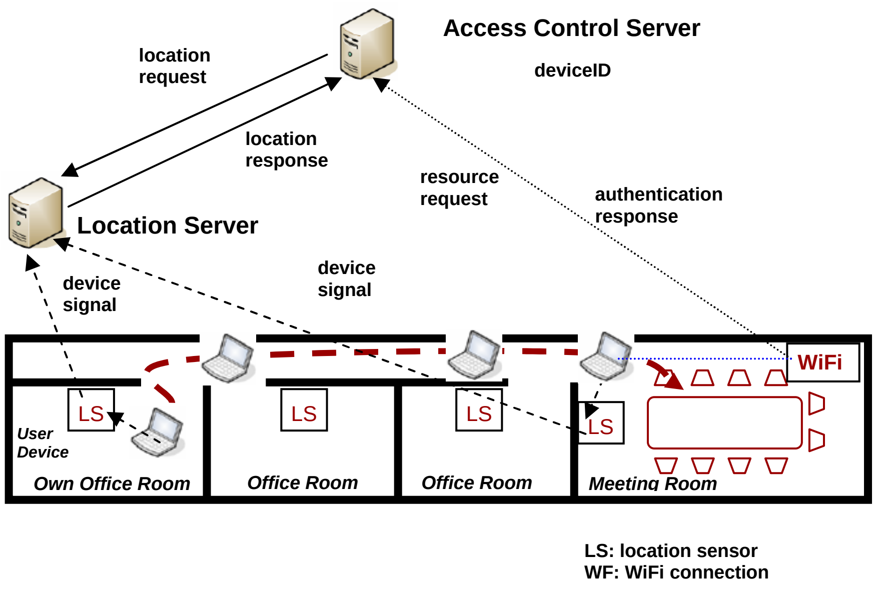

Fig. 13.1 Location Based Access Control System (based on [3])

The effectiveness of the access control solution of LBACS depends on several conditions regarding the operation of the different components that constitute it at runtime including, for example:

(C1) The continuous *availability* of the location servers and TPMs on the user devices at runtime. The availability of these components is a pre-requisite for the availability of device position and authentication information, which is necessary for the access control system at runtime.

(C2) The continuous periodic dispatch of signals from the mobile devices to the location server that enables it to maintain accurate position data for the devices.

Monitoring the above conditions at runtime in a system like LBACS would require the implementation of appropriate checks within the system itself or the deployment of an external monitor that would undertake the relevant responsibility. The former option would not be very flexible as it would require changes in the implementation of the required checks when the different components of the system change. Also depending on changes on the system components, the exact conditions that would need to be monitored could change as well. In such cases, giving the system the responsibility for monitoring would not be flexible.

The solution advocated in SERENITY is to delegate this responsibility to external components that would check the above conditions and take action when they are violated, e.g., replace malfunctioning components, alert system administrators of detected violations etc. In particular, in SERENITY the responsibility for monitoring runtime conditions is assigned to EVEREST and the responsibility for reacting to violations of properties is assigned to the SERENITY runtime framework. The generic architecture of EVEREST and its relation to the SERENITY runtime framework are discussed in the following.

13.3. Overview of EVEREST

The architecture of EVEREST is shown in Fig. 13.2. As shown in the figure, EVEREST is exposed as a service to the SERENITY runtime framework, offering interfaces for submitting monitoring rules to it for checking, forwarding runtime events from the applications which are being monitored, and obtaining monitoring results. Internally, EVEREST has three main components: a *monitor manager*, a *monitor* and an *event collector*.

The monitoring manager is the component that has responsibility for initiating, coordinating and reporting the results of the monitoring process. As such, it receives the monitoring rules from the SRF and provides the API for obtaining monitoring results. The event collector is responsible for receiving events from SRF and passing them to the monitoring manager. The monitoring manager forwards these events to the *Native Type Generator* (NTG) sub-component of the monitor, which translates the events from XML to internal Java objects. After receiving events from the manager, the monitor checks whether they violate any of the rules given to it.

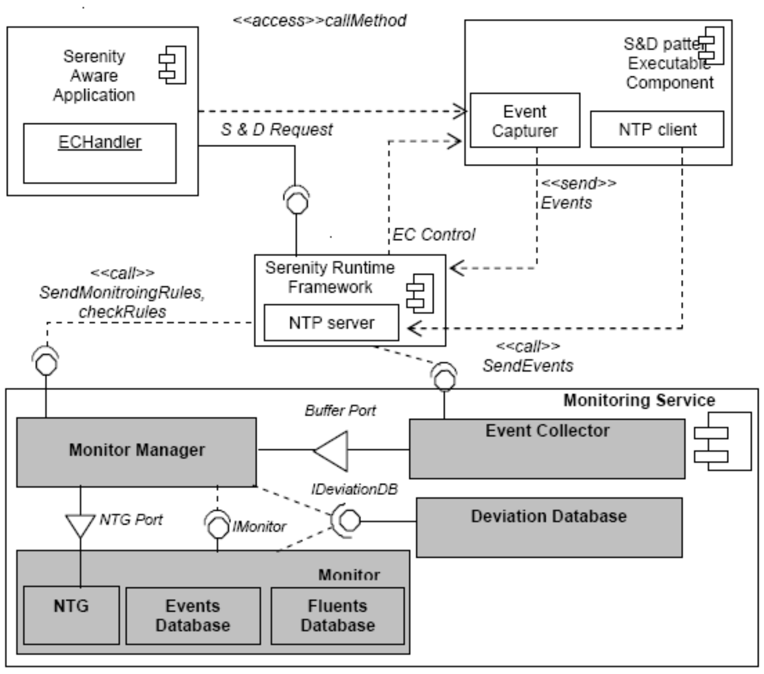

Fig. 13.2 Architecture of the monitoring framework

The monitor is a generic engine for checking violations of EC-Assertion formulae against a given set of runtime events. During monitoring, it also takes into account information about the state of a system, which it derives from runtime events using *assumptions*. To perform the required checks, the monitor maintains an *event* and a *fluent* database. The first of these databases keeps events which are necessary for checking past formulae (i.e., formulas requiring that when an event happens some other event should already have occurred or some condition should be satisfied), while the second keeps information about the initiation and termination of state conditions by runtime events that is necessary for monitoring (see Section 13.4). When a violation of a property is detected, the monitor records it in a *deviation database*. This database is accessed by the monitoring manager when the latter component is polled by the SRF to report detected deviations.

The *event capturers* intercept events during the operation of applications and send them to the SRF, which subsequently forwards them to EVEREST. Capturers are typically part of the implementation of the components that realise the solutions described by S&D Patterns. In some cases, however, they may also be part of the infrastructure where these components that realise S&D Patterns are deployed. When a capturer intercepts an event, it wraps it into an envelope containing additional information about the event. This information includes the *sender*, *receiver* and *source* of the event (i.e., the component where it was captured), and a *timestamp* indicating when the event was captured at its source.

As the event capturers may run on separate machines from the monitoring services, it is necessary to ensure that the timestamps of the events that they generate are comparable. To enable this, the event capturers that are provided by implementations of S&D Patterns should realise the *Network Time Protocol* [23], i.e., a protocol based on the clock synchronisation scheme described in [20]. The implementation of this protocol allows event capturers to compute the difference of their clocks with the clock of the SERENITY runtime framework at regular intervals. This difference is subsequently used to transform timestamps taken according to the clock of each capturer into timestamps that express time in terms of the SERENITY runtime framework's clock. This is achieved by implementing an *NTP client* at each event capturer and an *NTP server* at the machine that hosts the SERENITY runtime framework. The NTP clients call the NTP server at regular intervals to synchronise their clocks with the clock of the server. The use of NTP can synchronise distributed clocks at a very high level of accuracy since recent versions of NTP (e.g. version 4) use a resolution of less than one nanosecond.

13.4. Specification of Monitoring Rules and Assumptions in S&D Patterns

The rules that need to be monitored at runtime and other functional and non functional assumptions about the solutions which are being monitored are specified within S&D Patterns using an XML based language, called *EC-Assertion*. *EC-Assertion* is based on *event calculus* [27], a first-order temporal logic language that was originally developed to represent and reason about actions and their effects over time. The basic modelling constructs of Event Calculus are *events* and *fluents*. An event in EC is something that occurs at a specific instance of time, is of instantaneous duration, and may cause some change in the state of the reality that is being modelled. This state is represented by fluents.

To represent the occurrence of an event, EC uses the predicate *Happens(e, t, $\Re$(t1,t2))*. This predicate represents the occurrence of an event *e* that occurs at some time point *t* within the time range *$\Re$(t1, t2)* and is of instantaneous duration. The boundaries of *$\Re$(t1, t2)* can be specified by using either time constants or arithmetic expressions over the time variables of other predicates in an EC formula. The EC predicate *Initiates(e, f, t)* signifies that a fluent *f* starts to hold after the event *e* occurs at time t. The EC predicate *Terminates(e, f, t)* signifies that a fluent *f* ceases to hold after the event *e* occurs at time *t*. An EC formula may also use the predicates *Initially(f)* and *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.

EC defines a set of axioms that can be used to determine when a fluent holds based on initiation and termination events that regard this fluent. These axioms are listed in Table 13.1. Axiom *EC1* 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*. 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*. 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, axiom *EC4* states that the time range in a *Happens* predicate includes its boundaries.

Table 13.1. Axioms of Event Calculus

```
(EC1)   Clipped(t1,f,t2) ⇐ (∃e,t) Happens(e,t,ℜ(t1,t2))
                                  ∧ Terminates(e,f,t)
(EC2)   HoldsAt(f,t) ⇐ Initially(f) ∧ ¬Clipped(0,f,t)
(EC3)   HoldsAt(f,t) ⇐ (∃e,t1) Happens(e,t,ℜ(t1,t))
                               ∧ Initiates(e,f,t1)
                               ∧ ¬Clipped(t1,f,t)
(EC4)   Happens(e,t,ℜ(t1,t2)) ⇒ (t1 < t) ∧ (t ≤ t2)
```

EC-Assertion adopts the basic representation principles of EC and its axiomatic foundation and introduces special terms to represent the types of events and conditions that are needed for runtime monitoring. More specifically, given its focus on monitoring the operation of software systems at runtime, events in EC-Assertion can be invocations of system operations, responses from such operations, or exchanges of messages between different system components. To represent these types of events, *EC-Assertion* defines a specific event structure that is syntactically represented by the event term

event(_id, _sender, _receiver, _status, _sig, _source)

In this event term:

- *_id* is a unique identifier of the event;
- *_sender* is the identifier of the system component that sends the message/operation call/response;
- *_receiver* is the identifier of the system component that receives the message/operation call/response;
- *_status* is the processing status of an event (i.e., *REQ* if the event represents an operation invocation and *RES* if the event represents an operation response);
- *_sig* is the signature of the dispatched message or the operation invocation/response that is represented by the event, comprising the operation name and its arguments/result;
- *_source* is the identifier of the component where the event was captured.

Fluents are defined as relations between objects and represented as terms of the form $rel(O_1, \dots, O_n)$. In fluent terms, *rel* is the name of a relation which associates the objects $O_1, \dots, O_n$.

The rules to be monitored at runtime are specified in terms of the above predicates and have the general form *body* $\Rightarrow$ *head*. The meaning of a rule is that if its *body* evaluates to *True*, its *head* must also evaluate to *True*. The *Happens* predicates in a rule with no constraints for their lower and upper time boundaries are what we call “unconstrained” predicates. During the monitoring process, rules are activated by events that can be unified with the unconstrained *Happens* predicates in them. When this unification is possible, the monitor generates a rule instance to represent the partially unified rule and keeps this instance active until all the other predicates in it have been successfully unified with events and fluents of appropriate types or it is deduced that no further unifications are possible. In the latter case, the rule instance is deleted. When a rule instance is fully unified, the monitor checks if the particular instantiation that it expresses is satisfied.

Considering the location based access control scenario that we introduced in Section 13.2, the condition (C1) about the availability of location servers during the operation of LBACS can be checked by monitoring whether each time that the control server sends a request for the position of a specific device to the location server, the latter component responds to it within a predefined time interval, e.g., within 10 time units after the receipt of the request. This would be a *bounded availability* check which can be expressed in *EC-Assertion* by the following monitoring rule:

Rule-1:
```
  Happens(e(_e1, _controlServer, _locationServer, REQ
           location(_dev,_loc,_acc), _controlServer), t1, ℜ(t1,t1))
  ⇒ (∃ t2:Time, e2:String)
  Happens(e(_e2, _locationServer, _controlServer, RES,
           location(_dev,_loc,_acc),_controlServer),
           t2, ℜ(t1+1,t1+10))
```

The specification of *Rule-1* assumes that the operation of the location server providing the latest known position of a device is *location(_dev, _loc, _acc)*, where *_dev* identifies the device and *_loc*, *_acc* the location returned by the server and the estimation of its accuracy respectively.

Also the condition (C2) about the continuous periodic dispatch of signals from the mobile devices to the location server can be specified by the following two rules:

Rule-2:
```
  Happens(e(_e1, _dev, _locationServer, REQ, signal(_dev),
            _locationServer), t1, ℜ(t1,t1))
   ⇒ (∃ t2:Time, e2:String))
    Happens(e(_e2, _dev, _locationServer, REQ, signal(_dev),
              _locationServer), t2, ℜ(t1,t1+m)) ∧ (_e1 ≠ _e2)
```

Rule-3:
```
  Happens(e(_e1, _controlServer, _locationServer, REQ,
          location(_dev,_loc,_acc), _controlServer), t1, ℜ(t1,t1))
```

```
∧ ¬∃t. Happens(e(_e3, _controlServer, _locationServer, REQ,
          location(_dev,_loc,_acc), _controlServer), t, ℜ(0,t1))
⇒ (∃ t2:Time, e2:String)
Happens(e(_e2, _dev, _locationServer, RES, signal(_dev),
         _locationServer), t2, ℜ(t1,t1+m))
```

Rule-2 checks whether each mobile device (*_dev*), sends signals *signal(_dev)* periodically to the location server (*_locationServer*), with a maximum delay of up to *m* time units between two signals. A violation of this rule by a device would indicate that either the device malfunctions or that it is no longer present in the area covered by the system. *Rule-2* would be able to capture the latter case after a device becomes known to the system by sending it a signal for the first time but would not be able to capture cases where a known user with a malfunctioning device enters the area covered by the system. *Rule-3* above covers this case by checking whether the location server receives a signal from a device within a period of at most *m* time units after the first time that the control server makes a request for the location of this device.

13.5. Core Monitoring Capabilities

As discussed in Section 13.3, runtime events may come from distributed components operating with different time clocks. Furthermore, distributed system components may have different types of connections with the monitor and, therefore, generate events that arrive at EVEREST with different communication delays and possibly in an order that is not the same as the order of their generation.

Thus, EVEREST has to overcome two problems when checking properties involving events from distributed components: *(i)* to synchronise the clocks of the various event sources, so that the timestamps of the different events can be ordered and comparable to each other, and *(ii)* to establish until when a particular event needs to be stored, so that it can reason about the system properties in a sound way or, equivalently, to compute the required *monitoring lifetime* of each event.

Consider, for instance, the case where the system of Fig. 13.1 needs to be protected against attackers flooding the servers with false device signals. To detect such attacks, one possible condition to monitor is whether the signals sent to the location server have indeed been sent by the devices they appear to be coming from and that these devices have been authenticated to the system. This condition can be monitored using the following monitoring rule:

Rule-4:

```
Happens(e(_e1, _dev, _locationServer, REQ, signal(_dev),
        _locationServer), t1, ℜ(t1,t1))
```

```
 ⇒ (∃ t2:Time)
Happens(e(_e2, _dev, _locationServer, REQ, signal(_dev),
        _dev), t2, ℜ(0,t1))
 ∧ (∃ t3:Time)
Happens(e(_e3, _controlServer, _locationServer, REQ,
        location(_dev,_loc,_acc), _controlServer), t3, ℜ(0,t1))
```

In this rule, the predicate *Happens(e(_e1, _dev, _locationServer, REQ, signal(_dev),_locationServer), t1, ℜ(t1,t1))* represents the receipt of a signal from a device *_dev* by the location server and the predicate *Happens(e(_e2, _dev, _locationServer, REQ, signal(_dev), _dev), t2, ℜ(0,t1))* represents the dispatch of a matching signal from the same device which has occurred earlier. Also, the predicate *Happens(e(_e3, _controlServer, _locationServer, REQ, location(_dev,_loc,_acc), _controlServer), t3, ℜ(0,t1))* represents a request regarding the position of the particular device that has been issued by the control server of LBACS at some time point before the receipt of the device signal by the control server. The existence of such an earlier request indicates that the device is known to the system.

It should be noted that *Rule 4* tries to combine events from different sources, namely the location server (*_locationServer*), mobile devices (*_dev*) and control server (*_controlServer*) and these events may reach the monitor in an order that is different from the order of their creation. Thus, when the monitor receives the event *_e1* in the rule that represents a device signal captured at the location server, it will have to decide for how long it should wait for a correlated event *_e2* representing the same signal as captured at the device side, and wait for this event before deciding whether the rule has been violated. Otherwise, it may report a false violation of *Rule 4*. This would happen in cases where, after receiving *_e1*, the monitor receives events *_e2* and *_e3* corresponding to it.

The clock synchronisation, which is performed by the monitoring framework through the use of the *Network Time Protocol* (NTP), solves the first problem of how to synchronise the clocks of the different event sources but not the second, that is, the problem of estimating for how long events should be maintained to ensure the completeness of reasoning.

In the following, we present the mechanism that EVEREST uses for computing the lifetime of events received from distributed sources, along with the monitoring process that is realised by the framework

13.1.1. Computing the Lifetime of Events

Let us assume without loss of generality that *_dev*, *_locationServer* and *_dev* in *Rule 4* above denote both the source of the event and the *clock* of this source. As the occurrence of events of type $e_1^{locationServer}$ in *Rule 4* is unconstrained[1], events of this type can instantiate the rule during monitoring. Unlike them, events of type e_2^{dev} and $e_3^{controlServer}$ are temporally constrained by $e_1^{locationServer}$ events in the rule and cannot, therefore, create new instances of the rule; they can only be unified with existing rule instances.

Normally, if the monitor would receive an event of type $e_1^{locationServer}$ then it would create a new template of *Rule 4* for it and attempt to retrieve past e_2^{dev} and $e_3^{controlServer}$ events from the past event database to unify them with this template. If no such past events existed then it would report a violation. However, it is possible that such past events of type e_2^{dev} and $e_3^{controlServer}$ might have occurred but not received yet by the monitor due to communication delays. Thus, to be certain that the monitor does not report a false violation of Rule 4, the evaluation of the rule needs to be postponed until it is guaranteed that events of types e_2^{dev} and $e_3^{controlServer}$ cannot have occurred. Thus, there is a need to compute an upper time limit until which the monitor has to delay the evaluation of the rule's template to guarantee that no such events might have occurred but not received by it. This upper limit can be computed by examining the temporal constraints of the events in the rule – i.e., (1) $t2 \leq t1$ and (2) $t3 \leq t1$. It should be noted, however, that *t1*, *t2* and *t3* all refer to different clocks, i.e., the clocks of *_locationServer*, *_dev* and *_controlServer,* respectively.

In general, for a rule with *n+1 Happens* predicates, there will be at most *2n+1* such constraints (inequalities) to solve. This is because at least one of the rule predicates is unconstrained (needed for triggering the rule), the remaining *Happens* predicates contribute two inequalities each (one for the lower boundary of the time variable of the predicate and one for the upper boundary), and there will be an extra constraint (equality) establishing the exact value of the time variable of the event in question (i.e., the *t2* variable that is associated with the e_2^{dev} event in our example).

Fig. 13.3 presents the algorithm for computing the lifetime of an event. When an event *e* occurs, this algorithm first determines the set of rules *R(e)* which have predicates that can be unified with the event. This set includes rules that have event types which are the same as the type of *e* or super-types of it. Subsequently, the constraints of each rule in *R(e)* are identified and expanded with an equality expressing that the time variable of the predicate of the rule that has been unified with *e* is equal to the timestamp of *e* (step 2.a). Given the time constraint set that results from this process, the algorithm computes the maximum possible value for each of the time variables of the rule using the *Simplex* method [11] (step 2.b.i). By doing so for each rule, it effectively produces a set of constraints for the clocks of the various event sources, since the time variables refer to these clocks. It then groups the different time variables according to the clock of the event source they

[1] $e_1^{locationServer}$ abbreviates the event *e(e1, dev, locationServer, REQ, signal(dev), locationServer)*, where the subscript refers to the event ID and the superscript to the event source. Such abbreviated references are used in the rest of the chapter in all cases where other event variables are not important.

are related to (step 3), and generates a set of all the conditions, *Lifetime(e)*, for computing the upper bound of the lifetime of *e* (step 4). A condition in *Lifetime(e)* states that *e* will not be needed after the last event that is seen from a source/clock which is relevant to *e* has a timestamp, *last_observed*(c_j), that is greater than the maximum possible value of the time variables grouped in this clock's group, as expressed by the condition *last_observed*$(c_j)> max_{t_i \in G_j}(max(t_i))$. The reason for using the timestamp of the last event that has been observed from a clock in the evaluation of the *Lifetime(e)* conditions is because events are communicated from each source (event capturer) to the SERENITY runtime monitoring framework (and, therefore, to EVEREST) using TCP/IP protocol which *guarantee*s a FIFO transmission within the same source/SRF channel. The conditions in *Lifetime(e)* determine the lifetime of *e*, since the lifetime of *e* expires when their conjunction becomes true.

```
Compute_Lifetime(e):
1. R(e) = { r | r has a predicate p that can be unified with e}
2. Forall $r \in R(e)$ do
   a. $CN_r$= {time constraints of r} $\cup$ {time variable of predicate p that matches e =
      timestamp of e}
   b. Forall $t_i \in CN_r$ do
      i. Find $max(t_i)$ given $CN_r$
3. Group the time variables $t_i$ into as many groups $G_j$ as the different event sources
   (clocks) $c_j$ in R(e)
4. Lifetime(e) = $\cup_j$ ((last_observed($c_j$) > $max_{t_i \in G_j}(max(t_i))$))
```

Fig. 13.3 Computing the lifetime of an event – I

Assuming that *Rule 4* is the only rule being monitored and an event of type $e_1^{locationServer}$ is observed at t1=20, step 1 will produce the set $R(e_1^{locationServer})$ = *{Rule-4}*, step 2.a will produce $CN_r = \{t2 \leq t1, t3 \leq t1, t3 = 20\}$, step 2.b.i will produce the solutions *max(t1)= max(t2)= max(t3)=20* by finding the maximum value of *t1* for which the constraints in CN_r are satisfied, and step 3 will produce two groups of time variables *{t1}* and *{t2}*, for the two clocks *locationServer* and *dev*, respectively. Finally, in step 4, the lifetime constraint set for $e_1^{locationServer}$ will be established as:

$$Lifetime(e_1^{locationServer})=\{last\_observed^{controlServer}>20, last\_observed^{dev}>20, last\_observed^{locationServer}>20\}$$

The current implementation of the algorithm of Fig. 13.3 uses the Simplex method to find the maximum time of a time variable in *step 2.b.i.* Simplex has exponential complexity, $O(2^n)$, for a problem with *n* variables. Simplex has been chosen over algorithms with polynomial complexity (e.g., the worst case complexity of Karmarkar's algorithm [1] is $O(n^{3.5})$). This is because for small numbers of variables, as the ones normally appearing in monitoring rules ($n \leq 10$), Simplex has better performance. It should also be noted that the algorithm of Fig. 13.3

computes the maximum value of a time variable for each rule separately, rather than combining them into a single larger problem. This is because the individual rule problems can be solved independently and a larger set of rules would take more time to solve due to the additional time variables (since $2^n + 2^m < 2^{n+m}$ for $n,m \geq 2$). Due to this approach, once the individual rule inequality systems have been solved, the different time variables of events coming from the same clock need to be grouped together. This is done in step 3 of the algorithm.

Note also that the algorithm of Fig. 13.3 works under the assumption that the clocks/sources of the events in the rules are fully specified when a rule is matched with an incoming event. In the example of *Rule 4* this is the case, since all the sources are known. However, there might be cases where the exact source of events that could potentially be matched with a rule is not known after the rule is matched with arrived events. Consider, for instance, the following rule:

```
Rule-5:

  ∀ e1, e2, U: String; C1, C3: Terminal; C2: Component; t1, t2: Time
  Happens(e(_e1,_C1,_C2, REQ, login(_U,_C1), _C1),t1,ℜ(t1,t1))
  ∧ Happens(e(_e2,_C3,_C2, REQ, login(_U,_C3), _C3),t2,ℜ(t1,t2))
  ∧ _C1 ≠_C3 ⇒ ∃ e3: String; t3:Time
   Happens(e(_e3,_C1,_C2, REQ-A, logout(_U,_C1),_C1),t3,ℜ(t1+1,t2-1))
```

```
Compute_Lifetime(e):
1. $R(e) = \{ r \mid r$ has a predicate $p$ which unifies with $e\}$
2. Forall $r \in R(e)$ do
   a. $CN_r = \{$time constraints of $r\} \cup \{$ time variable of predicate $p$ that matches $e$ = timestamp of $e\}$
   b. Forall $t_i \in CN_r$ do
      i. Find $max(t_i)$ given $CN_r$
3. Group the time variables $t_i$ into as many group types $TG_u$ as the different types of event sources $c_u$ in $R(e)$
4. Forall group types $g \in TG_u$ do
   c. Forall the known sources $j$ of type $g$ do
      i. Create a group $G_j$ and assign copies of the time variables of $g$ to it
5. $Lifetime(e) = \cup_j\{(last\_observed(c_j) > max_{t_i \in G_j}(max(t_i)))\}$
```

Fig. 13.4 Computing the lifetime of an event – II

Rule 5 requires that if a user *U* logs in to a system *C2* from a terminal *C1* and later he/she logs in again from a different terminal *C3*, he/she must have logged out from the former terminal before the second login. The rule effectively monitors cases where users are logged in from different terminals at the same time. When an event *e(e2,…,C3)* (or e_2^{C3} in our abbreviated form) arrives at the monitor, its lifetime will need to be estimated in reference to the maximum possible values of time variables *t1* and *t3*. In this case, however, the algorithm of Fig. 13.3 does not work, since at step 3 it is not known which other terminals the user of

e_2^{C3} may be using or, equivalently, which source clocks should be associated with the time variables *t1* and *t3*.

To deal with such cases, the algorithm of Fig. 13.3 is extended as shown in Fig. 13.4. The extended algorithm initially groups time variables into groups corresponding to the types of the event sources that are associated with them in the rules. Then, for each of the source type groups, it finds all the sources of the particular type that are known to the system, creates different groups for them and assigns copies of the time variables of each source type to each of the source groups that were generated from the type. Thus, if it is known that the system being monitored with *Rule 5* has 3 terminals, the algorithm of Fig. 13.4 will create different variable groups for each of these terminals and assign copies of the time variables *t1* and *t2* to each of these groups.

Having computed the *Lifetime(e)* constraint set upon the arrival of an event *e* at runtime, we use it to compute a vector with the maximum time values for *e* with respect to the different clocks related to it. For the ongoing example of *Rule 4*, the vector of $e_1^{locationServer}$ would be <20, 20, 20>. The event and its vector are then stored in the database of the monitor. At that point, the monitor also checks if the lifetime of some previous event, which depends on the clock of the new event, has expired and removes all these events, if any. This process is shown in Fig. 13.5.

```
1. Observe an event e
2. Update the global vector of observed clock values
3. Lifetime(e) = Compute_Lifetime(e)
4. Store e in the DB with its vector of different clock limits
5. Remove events from the DB if their clock limits have been exceeded
```

Fig. 13.5 Algorithm for using event lifetimes

13.1.1. Monitoring Algorithm

To check for violations of monitoring rules, EVEREST maintains templates that represent different instantiations of the rules generated from the events sent to it at runtime. A template for a rule *r* stores:

- The identifier (*ID*) of *r*.
- A set of value bindings (VB) for the variables of the rule predicates that is generated from the unification of different events with these predicates.
- For each predicate *p* in *r* :
  - The *quantifier* of its time variable (Q) and its signature (SG).
  - The boundaries (*LB*, *UB*) of the time range within which *p* should occur.

– The *truth-value (V)* of *p*. V can be: *UN* if the truth value of the predicate is not known yet; *T* if the predicate is known to be true, or *F* if the predicate is known to be false.
– The *source* (SC) of the evidence for the truth value of *p*. The value of SC can be: *UN* if the truth value has not been established yet; *RE* if the truth value of the predicate has been established by a recorded event; or *NF* if the truth value of the predicate has been established by the principle of negation as failure.
– A *time stamp* (*TS*) indicating the time in which the truth-value of *p* was established.

EVEREST creates a set of *deviation templates* that represent instantiations of monitoring rules and are used to check for rule violations of rules.

These templates are updated by recorded and derived events. More specifically, when a new event *e* occurs, EVEREST identifies the templates that contain predicates which could be unified with *e* and templates having predicates whose truth value can be affected by the time indicated by *e* (e.g. predicates expected to be true by a specific time point which *e* shows that has passed) and updates them. The update can affect the variable binding of an identified template and/or the truth value of the predicates in it. This depends on the quantification of the time variable of each predicate.

Template-1							
ID	Rule 1						
VB	(_e1,?) (_e2,?) (_controlServer,?) (_locationServer,?) (_dev,?)(_loc,?)(_acc,?)						
P	**Q**	**SG**	**TS**	**LB**	**UB**	**TV**	**SC**
1	∀	**Happens**(e(_e1,_controlServer,_locationServer,REQ, location(_dev,_loc,_acc), _controServer),t1,ℜ(t1,t1))	t1	t1	t1	UN	UN
2	∃	**Happens**(e(_e2,_locationServer,_controlServer,RES, location(_dev,_loc,_acc),_locationServer),t2,ℜ(t1+1,t1+10))	t2	t1+1	t1+10	UN	UN

Fig. 13.6 Template for Rule 1

In particular, the truth value of a predicate of the form *$(\forall t)p(x,t)$* where *t* is unconstrained (i.e., it is defined to be in a range of the form *$\Re(t,t)$*) is set to *T(true)* as soon as an event that can be unified with *p* is encountered. The truth value of a predicate of the form *$(\forall t)p(x,t)$* where *t* is constrained to be in the range *$\Re(t1,t2)$* is set to *F (false)* as soon as an event which is not unifiable with *p* occurs between *t1* and *t2*, and to *T (true)* if all the events that occur at the distinguishable time points between *t1* and *t2* can be unified with *p*. The truth value of predicates of the form *$\neg(\forall t)p(x,t)$* where *t* must be in the range *$\Re(t1,t2)$* is set to *T (true)* as soon as the first event that is not unifiable with *p* occurs within the time range *$\Re(t1,t2,$* and *F(false)* if all the events at the distinguishable time points between *t1* and *t2* can be unified with *p*.

The truth value of a predicate of the form *$(\exists t)p(x,t)$* where *t* is in the range *$\Re(t1,t2)$* is set to *T (true)* as soon as the first event *e* that can be unified with *p* occurs between *t1* and *t2*. If no such event occurs within *ℜ(t1,t2)*, the truth value of *p* is set to *F (false)* by virtue of the principle of the *negation as failure* (NAF). The absence of events unifiable with *p* is confirmed as soon as the first event that cannot be unified with *p* occurs after *t2*. The truth value of a predicate of the form *¬(∃t)p(x,t)* is established in the opposite way: as soon as an event *e* that can be unified with *p* occurs between *t1* and *t2* the truth value of *p* is set to *F (false)* and if no such event occurs between *t1* and *t2*, the truth value of *p* is set to *T(true).*

As an example of this process consider the monitoring of *Rule 1*. Initially, the template for this rule will have no bindings for the time and non time variables of any of the predicates of the rule as shown in Fig. 13.6. Furthermore, the truth values of all the predicates in the template will be UN (unknown).

Then, assuming that an event *E1: Happens(e(id1, S1, R1, REQ, location(d1,l1,a1), S1)*, 24500) occurs, EVEREST will detect that *E1* can be unified with the first predicate in the template (i.e., the predicate *Happens(e(_e1,_controlServer,_locationServer,REQ,location(_dev,_loc,_acc),_contro Server),t1,ℜ(t1,t1))*) and create a new instance of the template in which E1 is unified with this predicate. Following the unification, the truth value (TV) of the predicate will be set to *T* and a new template representing the update will be created. This template is shown in Fig. 13.7. In the new template, the source (SC) of the truth value of the *Happens(e(_e1,…),t1,ℜ(t1,t1))*) will be set to RE (since the event that determined the truth value a recorded event), the timestamp at which the truth value of the predicate was determined will be set to 24500 (i.e., the timestamp of the event that was unified with the predicate) and the lower (LB) and upper (UB) time boundaries of the time variable of the predicate are both set to 24500.

Template-2							
ID	Rule 1						
VB	(e1,id1) (e2,?) (controlServer,R1) (locationServer,S1) (dev,d1)(loc,l1)(acc,a1)						
P	**Q**	**SG**	**TS**	**LB**	**UB**	**TV**	**SC**
1	∀	**Happens**(e(_e1,_controlServer,_locationServer,REQ, location(_dev,_loc,_acc), _controServer),t1,ℜ(t1,t1))	24500	24500	24500	T	RE
2	∃	**Happens**(e(_e2,_locationServer,_controlServer, RES,location(_dev,_loc,_acc),_locationServer),t2, ℜ(t1+1,t1+10))	t2	24501	24510	UN	UN

Fig. 13.7 Template for Rule 1 updated due to event E1

The update of the template due to the event *E1* will also change the variable binding (VB) of the template. More specifically, the variables *e1*, *server*, and *client* of the predicate *Happens(e(e1,…),t1,ℜ(t1,t1))* will be bound to the values *id1*, *R1*, and *S1* respectively. Furthermore, the update will affect the lower boundary (LB) upper boundary (UB) of *t2,* i.e., the time variable of the predicate *Hap-*

pens(e(_e2,_locationServer,_controlServer,RES,location(_dev,_loc,_acc),_locatio nServer),t2,ℜ(t1+1,t1+10)) in the template. This is because the boundaries of *t2* depend on the value of the time variable *t1* that has been changed (set) by the update. In particular, the lower and upper boundary of *t2* will be set to 24501 (i.e., *t1+1*) and 24510 (i.e., *t1+10*) respectively.

Subsequently, if an event *E2: Happens(e(id2, S1, R1, REQ, authorise(), R1), 24507)* occurs at the time point t=24507, the template of Fig. 13.7 will be updated again. This is because *E2* can be unified with the predicate *Happens(e(_e2,_locationServer,_controlServer,RES,location(_dev,_loc,_acc),_locatio nServer),t2,ℜ(t1+1,t1+10))* in the template and has occurred within the time boundaries of this predicate (i.e., between 24501 and 24510). The result of this update is shown in Fig. 13.8. As shown in the figure, the truth value of the predicate *Happens(e(_e2,_...),t2,ℜ(t1+1,t1+10))* is set to true (*T*), its timestamp is set to 24507 and the source of the truth value of the predicate is set to RE as E2 was also a recorded event.

Template-2							
ID	Rule 1						
VB	(e1,id1) (e2,id2) (controlServer,R1) (locationServer,S1) (dev,d1)(loc,l1)(acc,a1)						
P	**Q**	**SG**	**TS**	**LB**	**UB**	**TV**	**SC**
1	∀	**Happens**(e(_e1,_controlServer,_locationServer,REQ, location(_dev,_loc,_acc), _controServer),t1,ℜ(t1,t1))	24500	24500	24500	T	RE
2	∃	**Happens**(e(_e2,_locationServer,_controlServer,RES, location(_dev,_loc,_acc), _locationServer),t2,ℜ(t1+1,t1+10))	24507	24501	24510	T	RE

Fig. 13.8 Template for Rule 1 as updated following events E1 and E2

Once the truth values of all the predicates in a template have been determined, the template is checked for violations. At this point if the truth value of all the predicates in the body of the template is *true* and the truth value of at least one predicate in the head is *false* then the instance of the rule represented by the template is violated. Otherwise, the template is satisfied.

The monitoring process described above is followed in cases of rules like *Rule 1* which are *future* EC-Assertion formulas (i.e., formulas in which the unconstrained time variable of the rule can only take values which are less than the values of the constrained time variables) and assuming that the events arrive at the monitor in the exact order of their occurrence. A monitoring rule, however, can also be a *past* formula, i.e., a formula having at least one constrained time variable that is constrained to take values which are less than or equal to the value of the unconstrained time variable of the formula. An example of a *past* monitoring rule is *Rule 4*.

In this rule, the unconstrained time variable is *t1* (since its boundaries are defined without reference to other time variables) and the constrained time variables are *t2* and *t3*. A rule of this type is checked by a different procedure. More specifically, if EVEREST receives an event which can be unified with a constrained predicate in a rule whose unconstrained predicate has not been instantiated yet (e.g. an event that could be unified with the predicates *Happens(e(_e2, _dev, _locationServer, REQ, signal(_dev), _dev), t2, $\Re$(0,t1))* or *Happens(e(_e3, _controlServer, _locationServer, REQ, location(_dev,_loc,_acc), _controlServer), t3, $\Re$(0,t1))* in Rule 4), it stores the event in a database of past events, shown as "Events Database" in Fig. 13.2 but does not instantiate the template. Later, when EVEREST receives an event which can be unified with the unconstrained predicate of the rule, it proceeds with the creation of a new template and then searches the past events database to check if there are already events which could currently be unified.

A similar approach is applied for *HoldsAt* predicates in rules, since according to the EC axioms shown in Table 13.1, *HoldsAt* is a derived predicate whose truth value depends on the existence of past *Initiates* and *Terminates* predicates. These predicates are derived from the assumptions of a theory, which state what events initiate and respectively terminate a particular fluent. To check the truth values of *HoldsAt* predicates, EVEREST stores *Initiates* and *Terminates* predicates into *its-Fluent Database* (see Fig. 13.2) and when it needs to evaluate a *HoldsAt* at some future time instance *t1*, it searches this database for the most recent *Initiates* and *Terminates* predicates which precede *t1* and satisfy the axioms in Table 13.1 for *HoldsAt* predicates.

13.6. Implementation and Evaluation

EVEREST has been implemented in Java and can be deployed either through the SRF or as a standalone web service. The implementation of EVEREST has been evaluated in a series of experiments that have focused on the performance of the core monitoring process that is realised by the framework and the effect that it has on the performance of the systems that it monitors. A detailed account of this evaluation is beyond the scope of this chapter and may be found in [17]. In the following, however, we summarise the main findings of the evaluation experiments of the framework to enable a better understanding of its capabilities and limitations.

More specifically, the evaluation of EVEREST has demonstrated that in the general case the time required to detect violations of monitoring rules after all the events that would enable this become available, increases exponentially with the number of the events that are sent to the monitor.

The violation detection time depends on the number of active templates when the last event that enables making a decision about the violation or not of a rule becomes available. The latter number depends on the exact form of the rules that are being monitored and, thus, it may be reduced substantially for specific types of rules. For example, in the case of rules that express typical security properties, notably *confidentiality*, *integrity* and *availability,* the number of active templates and, consequently, the violation detection time increases linearly with the number of events, as discussed in [17]. This is because confidentiality and integrity properties are expressed by past *EC-Assertion* rules as it has been shown in [29]. As discussed earlier, past rules are of the form *Happens(e1,t1,R(t1,t1))* $\Rightarrow$ *Happens(e2,t2,R(0,t1))* and therefore when the event(s) that satisfy the conditions in the body of the rule (e1) occur(s), the event(s) in the head of the rule (e2) must have occurred already. Thus, the monitor has only to check whether other events have taken place previously or certain conditions hold. Consequently, in such cases there is no need for maintaining partially instantiated instances of rules (templates) and wait for future events that could be unified with these instances, something that would add a considerable computational cost to the monitoring process. Also in the case of bounded availability rules (as *Rule-1* in this chapter), the key factor for performance is the period within which a response is expected following a request. As in most cases of synchronous communication the acceptable delay for a response is very low, the use of bounded availability rules with short waiting periods does not affect the performance of the monitor significantly, as observed in [17].

Furthermore, the evaluation in [17] and evaluations of predecessors of EVEREST [19, 30] have demonstrated that the performance of the monitor is not affected significantly by the use of assumptions and the subsequent deployment of the deductive reasoning capability of the toolkit in order to deduce information from these assumptions. The reason for this is that in typical monitoring scenarios, the number of successive deductive steps which are required in order to derive the information required from monitoring assumptions is very small (1 or 2 steps) and, therefore, the computational overhead of deductions during monitoring is also small.

Finally, previously conducted experiments have indicated that the performance of the monitor is not significantly affected by the size of the domains of the variables used in monitoring rules. Also the evaluation in [17] has indicated that the overhead of event capturing on the performance of the system that is being monitored depends on the type of the deployed capturer. This overhead ranges from a 18%–20% drop in performance, when events are captured from the execution platform of the application, to 800%, in cases where event capturers are implemented as wrappers of components of the system that is being monitored [17].

13.7. Related Work

Dynamic verification enables a software system to improve its dependability (and therefore security) [4], by checking whether its behaviour satisfies specific dependability and security properties while it is running. Dynamic system verification has emerged more recently and has been investigated in the context of different areas including requirements engineering, program verification, safety critical systems and service centric systems.

In requirements engineering, dynamic verification has focused on system requirements and investigated: *(i)* ways of specifying requirements for monitoring and transforming them into events that can be monitored at run-time; *(ii)* the development of event-monitoring mechanisms; *(iii)* the development of mechanisms for generating system events that can be used in monitoring (e.g., instrumentation, use of reflection [6]); and *(iv)* the development of mechanisms for adapting systems so as to deal with deviations from requirements at run-time as, for example, in [34].

In dynamic program verification, research has focused on the development of programming platforms with generic monitoring capabilities including support for generating program events at run-time, e.g., jMonitor [8], embedding specifications of monitoring properties into programs, and producing code that can verify these properties during the execution of the programs, e.g., monitoring-oriented programming [8]. The Java PathExplorer (JPaX) is a tool for monitoring systems at their runtime [14]. The use of JPaX enables the automatic instrumentation of code and observation of its runtime behaviour. JPaX can be used during development to provide more robust verification. It can also be used in an operational setting, to help optimize & maintain systems as they mature. In [15,16], a framework for evolvable software systems is proposed, based on runtime verification. In this framework components are considered as supervisors (monitor) and supervisees (evolvable component), where supervisor is the process that monitors and may evolve the supervisee. The supervisor maintains a meta-level theory for the object level of the supervisee, where the theory is specified in *revision based logic*. Meta level states are able to record observations of the supervisor's computational state and as well as the observations at the object level. The meta level and the object level states must be in accord. Thus, any revision action in the meta level that transforms the state of the supervisor may induce an accompanying transformation of the object level through reflection.

In service-centric systems, i.e., systems that deploy autonomous web services [18], the interest in dynamic verification has emerged due to the need to specify and monitor *service level agreements* between the providers and consumers of web-services being deployed in service-centric systems. As a result of recognizing the importance of this form of verification, work in this area has focused on the development of standards and languages for specifying monitoring properties and methods for monitoring them [5,18,26]. Dynamic verification has also focused on monitoring service level agreements (SLAs) [12,21]. In [24] a framework is presented to allow non-intrusive adaptation of partner services within a BPEL process, without any down time of the overall system. In this approach a BPEL process is monitored according to certain QoS criteria and existing partner services may be replaced (in case a partner fails to satisfy QoS criteria) based on various replacement strategies. The replacement service can either be syntactically or semantically equivalent to the interface used in BPEL.

Research on dynamic verification has also focused on system security. Work in this area has mainly been concerned with the development of *Intrusion Detection Systems (IDS)* [10] that use dynamic verification techniques for detecting security threats. In the literature, *IDSs* are classified based on different criteria. For example based on the source of the input to the IDS, these systems are classified as *Host Based IDS* and *Network Based IDS* [2]. Host Based IDS are mostly concerned with the examination of system logs of one or more application hosts [31, 32]. On the other hand, Network Based IDSs perform protocol analysis and content searching/matching on network traffic. These systems are commonly used to actively block or passively detect a variety of attacks and probes on IP networks [13, 28]. IDSs have also been distinguished into *centralized* and *distributed* systems depending on the form of intrusions that they focus: in centralized IDSs, intrusion detection occurs in a single monitored system [13,32], while in distributed IDSs, intrusion detection is performed across multiple network sites [7,8,25].

In comparison with the monitoring platforms overviewed above, EVEREST provides a more comprehensive monitoring framework as it can be applied not only to systems implemented in a specific programming language (e.g. Java), supports the specification of a wide range of monitoring rules with precise time constraints, and can deal with events that may be captured and notified from distributed sources and through different communication channels. Furthermore, EVEREST can support the monitoring of conditions at various levels (e.g. network and application levels).

13.8. Conclusions

This chapter has discussed the core monitoring capabilities that are available in the SERENITY runtime framework. These capabilities are offered by a generic runtime monitoring toolkit called EVEREST that can detect violations of properties expressed as monitoring rules in *EC-Assertion* – a formal temporal logic language that is based on Event Calculus.

Monitoring in SERENITY is activated when an S&D Pattern is selected and the SERENITY runtime framework activates a specific implementation of it. At this point, the SERENITY runtime framework extracts the monitoring rules specified within the pattern and submits them to EVEREST for monitoring. EVEREST subsequently checks these rules against events that are captured by event capturers associated with the active implementation of the pattern and sent to EVEREST via the SERENITY runtime framework.

EVEREST provides comprehensive monitoring support, enabling checks of monitoring rules that are expressed as past or future EC-Assertion formulas and against events that might have been captured by distributed event capturers. The toolkit has been implemented in Java and evaluated in a series of experiments with positive results.

Current work on EVEREST focuses on the expansion of its core monitoring capabilities to provide support for the detection of potential violations of monitoring rules (aka *threats*). This work is further discussed in [33]. Another area of investigation concerns the scope for possible optimisations of the reasoning process of EVEREST and, in particular, ways for distributing the checking of rules.

References

1. Adler I et al (1989) An Implementation of Karmarkar's Algorithm for Linear Programming. Mathematical Programming, 44: 297–335
2. Lazarevic A, Kumar V, Srivastava J (2006) Intrusion Detection: A Survey. Massive Computing, In: Kumar V, Srivastava J, Lazarevic A (eds), Managing Cyber Threats: Issues, Approaches and Challenges, Springer, ISBN 0387242260
3. Armenteros A, Garcia L, Muoz A, Maña A (2008) Realising the Potential of SERENITY in Emerging AmI Ecosystems: Implications and Challenges. In: Spanoudakis G, Maña A, Kokolakis S (eds) Security and Dependability for Ambient Intelligence, Information Security Series, Springer
4. Avizienis A, Larpie C, Randell B (2001). Fundamental Concepts of Dependability. LAAS-CNRS, Tech. Rep. N01145.
5. Baresi L, Guinea S (2005) Dynamo: Dynamic Monitoring of WS-BPEL Processes. Proceedings of 3rd International Conference On Service Oriented Computing, Amsterdam, The Netherlands.
6. Campbell A, Safavi-Naini R, Pleasants A (1992) Partial Belief and Probabilistic Reasoning in the Analysis of Secure Protocols. Proceedings of 5th IEEE Computer Security Foundations Workshop, 84-91. IEEE Computer Society Press.
7. Chatzigiannakis V, Androulidakis G, Grammatikou M, Maglaris B (2004) A Distributed Intrusion Detection Prototype using Security Agents. Proceedings of HP Open View University Association (HPOVUA)
8. Chatzigiannakis V, Androulidakis G, Grammatikou M, Maglaris B (2004) An Architectural Framework for Distributed Intrusion Detection using Smart Agents. Proceedings of SAM04, Las Vegas
9. Chen F, Rosu G (2003) Towards Monitoring-Oriented Programming: A Paradigm Combining Specification and Implementation. In Electronic Notes in Theoretical Computer Science, 89(2), Elsevier Science B.V.
10. Denning D (1987) An Intrusion-Detection Model. IEEE Transactions on Software Engineering, 13(2): 222-232.
11. Gale D (2007) Linear programming and the simplex method. Notices of the AMS, 54(3):364–369`.

12. Ghezzi C, Guinea S (2007) Runtime Monitoring in Service Oriented Architectures. In: Baresi L and di Nitto E. (eds), Test and Analysis of Web Services, Springer, 237-264, 2007.
13. Gudkov V, Johnson J (2002) Multidimensional Network Monitoring for Intrusion Detection. CoRR: Cryptography and Security/0206020
14. Havelund K, Roşu G (2004) An Overview of the Runtime Verification Tool Java PathExplorer. Form. Methods Syst. Des. 24, 189-215.
15. Barringer H, Rydeheard D, Gabbay D (2007) A Logical Framework for Monitoring and Evolving Software Components. Proceedings of 1st Joint IEEE/IFIP Symposium on Theoretical Aspects of Computer Science (TASE07), Shanghai.
16. Howard B, Dov G, Rydeheard D, (2007) From Runtime Verification to Evolvable Systems. 7th International Workshop on Runtime Verification
17. Kloukinas C, Mahbub K, Spanoudakis G (2007) Evaluation of V1 of Dynamic Validation Prototype, Deliverable A4.D3.2, SERENITY Project, http://www.serenity-forum.org/IMG/pdf/A4.D3.2_Evaluation_of_v1_of_dynamic_validation_prototype_v.-2.pdf, Accessed 9 December 2008
18. Mahbub K, Spanoudakis G. (2004) A Framework for Requirements Monitoring of Service Based Systems. Proceedings of 2nd International Conference on Service Oriented Computing, NY, USA.
19. Mahbub K, Spanoudakis G. (2005) Run-time Monitoring of Requirements for Systems Composed of Web-Services: Initial Implementation and Evaluation Experience. Proceedings of 3rd Int. IEEE Conf. on Web Services
20. Mahbub K, Spanoudakis G, Kloukinas C, (2007). V2 of dynamic validation prototype". Deliverable A4.D3.3, SERENITY Project, http://www.serenity-forum.org/IMG/pdf/A4.D3.3_-_V2_of_Dynamic_validation_Prototype.pdf. Accessed 9 December 2008
21. Mahbub K, Spanoudakis G (2007) Monitoring WS-Agreements: An Event Calculus Based Approach. In: Baresi L, and di Nitto E (eds), Test and Analysis of Web Services, Springer
22. Maña A et al (2006) Security engineering for ambient intelligence: A manifesto. In: Integrating Security and Software Engineering: Advances and Future Vision. Idea Group Publishing, 244–270
23. NTP, www.ntp.org, Accessed on 9 December 2008
24. Moser O, Rosenberg F, Dustdar S (2008) Non-intrusive monitoring and service adaptation for WS-BPEL. Proceedings of 17th International Conference on World Wide Web
25. Zhang Q, Janakiraman R (2001) Indra: A Distributed Approach to Network Intrusion Detection and Prevention. Washington University Technical Report # WUCS-01-30
26. Li Q (2007) A Dynamic Verification Platform for BPEL Environments. MSc. Thesis, Department of Electrical & Computer Engineering, University of Alberta
27. Shanahan M.P. (1999) The event calculus explained. In: Artificial Intelligence Today. Volume 1600 of Lecture Notes in Artificial Intelligence. (1999) 409–430
28. SNORT Intrusion Detection System, www.snort.org, 2004. Accessed 9 December 2008
29. Spanoudakis G, Kloukinas C, Androutsopoulos K.(2007) Towards security monitoring patterns. Proceedings of ACM Symposium on Applied Computing (SAC07) - Track on Software Verification, Volume 2, Seoul, Korea, 1518–1525
30. Spanoudakis G, Mahbub K (2006) Non intrusive monitoring of service based systems. Int. J. of Cooperative Information Systems 15: 325–358
31. Staniford-Chen S, Tung B, Porras P, Kahn C, Schnackenberg D, Feiertag R, Stillman M (1998) The Common Intrusion Detection Framework - Data Formats. IETF, www.watersprings.org/pub/id/ draft-staniford-cidf-data-formats-00.txt, Accessed on 9 December 2008
32. Stephen E, Hansen, E, Atkins T (1993) Automated System Monitoring and Notification With Swatch. Proceedings of 7th USENIX conference on System administration, Monterey, California, USA, 1993
33. Tsigritis T, Spanoudakis G, Kloukinas C, Lorenzoli D (2009) Diagnosis and Threat Detection Capabilities of the SERENITY Monitoring Framework. In Spanoudakis G, Maña A,

and Kokolakis S (eds), Security and Dependability for Ambient Intelligence, Information Security Series, Springer
34. van Lamsweerde A (1996) Divergent Views in Goal-Driven Requirements Engineering. Proceedings of Viewpoints '96 – ACM SIGSOFT Workshop of Viewpoints in Software Development