Towards Security Monitoring Patterns

George Spanoudakis
Department of Computing,
City University,
Northampton Square,
London, EC1V 0HB, U.K.
gespan@soi.city.ac.uk

Christos Kloukinas
Department of Computing,
City University,
Northampton Square,
London, EC1V 0HB, U.K.
C.Kloukinas@soi.city.ac.uk

Kelly Androutsopoulos
Department of Computing,
City University,
Northampton Square,
London, EC1V 0HB, U.K.
sbbb530@soi.city.ac.uk

ABSTRACT

Runtime monitoring is performed during system execution to detect whether the system's behaviour deviates from that described by requirements. To support this activity we have developed a monitoring framework that expresses the requirements to be monitored in event calculus – a formal temporal first order language. Following an investigation of how this framework could be used to monitor security requirements, in this paper we propose patterns for expressing three basic types of such requirements, namely confidentiality, integrity and availability. These patterns aim to ease the task of specifying confidentiality, integrity and availability requirements in monitorable forms by non-expert users. The paper illustrates the use of these patterns using examples of an industrial case study.

Categories and Subject Descriptors

D.2.4 [Software/Program Verification]: *assertion checkers.*

Keywords

Runtime monitoring, security patterns, event calculus

1. INTRODUCTION

Researchers and organizations are constantly developing new security mechanisms for deterring attackers, such as firewalls, virus detection systems and cryptographic protocols for secure communication. When developing a system with security mechanisms, they usually apply static analysis techniques in order to verify that the system behaviour respects certain security properties and detect flaws. Static analysis techniques range from static verification (e.g. model checking) to techniques measuring the efficiency of algorithms (e.g. encryption).

However, attackers are also constantly working on attack methods and hence a system can never be considered to be completely secure. Furthermore, static analysis techniques typically verify properties based on assumptions about the behaviour of a system that may turn out to be incorrect at runtime or models which may be incomplete. Thus, no matter how rigorous static analysis might have been, there is no guarantee that the final implementation will not be vulnerable to attack. To alleviate this problem, monitoring of security requirements can be applied to a system to detect at runtime whether its behaviour deviates from that described by its security requirements and the assumptions under which it was shown to be secure. Runtime monitoring can be used as a replacement of static verification or in addition to it. Runtime monitoring takes as input security requirements and checks whether they are consistent with traces of events that are produced during system execution.

In this paper, we introduce patterns for expressing basic security monitoring properties that can be checked at runtime using a general runtime requirements monitoring framework that is discussed in [26]. In this framework, requirements are expressed in *event calculus* (i.e., a first-order temporal formal language [24] shortly referred to as "EC" henceforth) in terms of events which signify the emission/reception of messages by different components of a system, and fluents that represent changes in the state of a system which are triggered by events. As correct EC formulas, however, can be difficult to write, non expert users could benefit from having abstract EC formulas (patterns) expressing generic security properties that could be instantiated to specify the exact security requirements that need to be monitored in specific systems. To enable this, we introduce patterns which cover the three main security properties as defined in [3], namely: (i) *Confidentiality* – the absence of unauthorised disclosure of information; (ii) *Integrity* – the absence of unauthorised transformations of the state of a system; and (iii) *Availability* – the readiness of a system to provide a correct service. Our work was initially motivated by the work on specification patterns [10][1] defined to assist users in expressing requirements in formal languages, such as LTL [20] and CTL [6]. However, these specification patterns have not been defined for event calculus nor do they focus on security requirements.

Using the patterns that we introduce in this paper, someone can implement basic security requirements of a system separately from the system functionality by creating security-related rules and verifying and controlling these requirements using the EC monitoring framework discussed in [26]. In this way it is possible to separate the treatment of the security requirements from the application logic. If the system already has built-in security mechanisms, then the external monitoring of security requirements adds yet another layer of security checking which is independent from the system and therefore makes it more fault tolerant [5], especially if it is possible to control the system when violations are detected. However, even in cases where control cannot be applied, our approach can help by providing the basis for detecting more violations than the system itself, logging violations, and using the sophisticated reasoning capabilities of the monitor to analyse specific dynamic event patterns, e.g., for cases like denial of service attacks.

SAC'07, March 11-15, 2007, Seoul, Korea.
Copyright 2007 ACM 1-59593-480-4 /07/0003…$5.00.

The rest of this paper is structured as follows. In Section 2, we describe the general framework that we use for monitoring and in Section 3 we overview event calculus. In Section 4, we define the monitoring patterns and present examples of using them to express security requirements drawn from an industrial scenario. In Section 5, we discuss related work and in Section 6 we give conclusions and present plans for future work.

2. MONITORING FRAMEWORK

The general architecture of the monitoring framework that we use to monitor the patterns introduced in this paper is shown in Figure 1. This framework consists of a monitoring manager that accepts a set of (security) requirements to be monitored and based on them it identifies the event types that need to be observed and sends them to an event catcher. It also forwards the requirements to be monitored to the monitoring engine of the framework.

The event catcher observes events from one or more of the agents that constitute the system being monitored, captures those that correspond to the event types given by the monitoring manager and sends them to the monitoring engine of the framework. The monitoring engine checks whether the event trace that is reported to it is consistent with the requirements and if it is not it reports a violation to the manager. It can also deduce information about the state of the system being monitored from the recorded events, using formulas that specify how events affect the state of the monitored system called *assumptions* (see Section 4.1), and the standard axioms of EC [24].

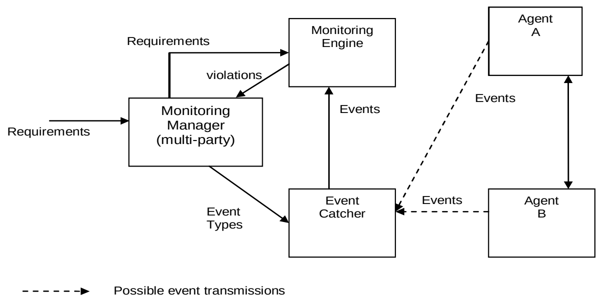

Figure 1. Monitoring Framework

Monitored systems may include one or more different interacting agents and different types of monitoring may be distinguished based on the capability to capture securely reliable runtime events from the different agents that constitute them. According to this criterion, monitoring can be distinguished into:

- *Single-party monitoring*: In this type of monitoring only one of the agents which are involved in an interaction is capable or willing to provide secure and reliable information about runtime events. This type of monitoring may be further distinguished, based on a directed communication from a source agent to a destination agent into source-party or destination-party monitoring.
- *Multi-party monitoring*: In this type of monitoring multiple agents in an interaction can provide secure and reliable information about runtime events.

Secure event reporting in this framework is based on a publish/subscribe architecture based on PKI [27].

3. SPECIFICATION OF PROPERTIES I

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

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

The occurrence of an event in EC is represented by the predicate *Happens*$(e,t,\Re(t_1,t_2))$. This predicate signifies that an instantaneous event *e* occurs at some time *t* within the time range $\Re(t_1,t_2)$. The boundaries of $\Re(t_1,t_2)$ can be specified by using either time constants or arithmetic expressions over the time variables of other predicates in an EC formula. The initiation of a fluent is signified by the EC predicate *Initiates(e,f,t)* whose meaning is that a fluent *f* starts to hold after the event *e* at time *t.* The termination of a fluent is signified by the EC predicate *Terminates(e,f,t)* whose meaning is that a fluent *f* ceases to hold after the event *e* occurs at time *t*. An EC formula may also use the predicates *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.

Our EC based language uses special types of events and fluents to specify monitorable properties of systems. More specifically, fluents can be defined by the user as relations between objects of the following general form:

$$relation(Object_1, \ldots, Object_n) \quad \text{(I)}$$

In (I), *relation* is the name of the relation that takes as arguments *n* objects ($\text{Object}_1, \ldots, \text{Object}_n$) that can be fluents or terms. A pre-defined relation for fluents that is commonly used is:

$$valueOf(variable,\ value\_exp) \quad \text{(II)}$$

whose meaning is that *variable* has the value *value_exp*. In (II),

- *variable* denotes a typed variable which can be:
  - (i) a *system variable* – A system variable is a variable of the system that is being monitored whose value can be captured at any time during the monitoring process, or
  - (ii) a *monitoring variable* – A monitoring variable is introduced by the users of the monitoring framework to represent the deduced states of the system at runtime (i.e. states which the system itself might not be aware of but its monitor can use in order to reason about it).
- *value_exp* is a term that either represents an EC variable/value or signifies a call to an operation that returns an object of the same type as the variable. This operation may be a built-in operation of the monitoring engine (e.g. an operation that computes the average of a set of values) or an operation that is invoked in an external party. When *value_exp* is an operation call, then effectively the return value of the operation becomes the value of *variable.*

Events in our framework represent exchanges of messages between the agents that constitute a system. A message can invoke an operation in an agent or return results following the execution of an operation. In our EC-based language, events are described by terms that have the following generic form:

event(_id, _sender, _receiver, _status, _oper, _source) (III)

In (III):

- *_ID* is a unique identifier of the event
- *_sender* is the identifier of the agent that sends the message.
- *_receiver* is the identifier of the agent that receives the message.
- *_status* represents the processing status of an event. The status of the event can be: (i) REQ-B, that is a request for the invocation of an operation that has been received but whose processing has not started yet; (ii) REQ-A, that is a request for the invocation of an operation that has been received and whose processing has started; (iii) RES-B, that is a response generated upon the completion of an operation that has not been dispatched yet; or (iv) RES-A, that is a response generated upon the completion of an operation that has been dispatched.
- *_oper* is the signature of operation that the event invokes or reports the results of.
- *_source* is the name of the agent that provided information about the event.

As an example of a monitoring property expressed in our EC-based language consider the formula below:

```
∀  _id1,_id2,_s,_r:String;_v:ObjType t:Time
  Happens(e(_id1,_s,_r,REQ-B, o(_v),_r), t1,
  ℜ(t1,t1)) ⇒ Happens(e(_id2,_r,_s,RES-A,
  o(_v),_r), t2, ℜ(t1,t1+tu))
```

According to this formula, an agent *_r* which receives an event invoking the operation *o(_v)* in it (i.e. *e(_id1,_s,_r,REQ-B, o(_v),_r)*) should complete the execution of *o(_v)* and respond to the caller (*_s*) within t_u time units following the request (i.e., an availability requirement as we discsuss in Section 4.4).

4. SECURITY MONITORING PATTERNS

The monitoring patterns that we have defined to enable the specification of monitoring rules focus on common security properties, namely confidentiality, integrity and availability. These patterns are introduced in the following after an overview of the monitoring pattern language that we use to express them.

4.1 Monitoring Pattern Language

A monitoring pattern is composed of: (i) a *monitoring rule* which defines in a parameterised form the event calculus formulas that will need to be monitored at runtime, and (ii) a set of *assumptions* which define in parameterised forms the event calculus formulas that can be used at runtime to deduce information about the state of the monitored systems that affects the satisfiability of the monitoring rules based on captured runtime events. Patterns may define different monitoring rules and assumptions for different types of monitoring (e.g. single and multi-party monitoring) if these types can be applied for the property of the pattern.

The first order language that is used to define the pattern formulas is based on EC. The variables of this language are typed and might or might not be replaced depending on their specification in the pattern. More specifically a variable *_x* which appears in "<>" (i.e. <_x>) must be replaced by another compatible variable, that is a variable whose type is compatible with the type of *_x* when the pattern is instantiated. Variables which do not appear in "<>" should not be replaced.

The type of a pattern variable may be one of the following predefined set of meta-types: *Term*, *Event*, *Fluent*, *Agent*, and *InformationTerm*. *Term* is the most general of these meta-types and represents any type of term that can appear in an EC formula. The meta-types *Event* and *Fluent* represent events and fluents respectively as defined in Section 3. The meta-type *Agent* represents an entity with a computational capacity that can send, receive and process messages. Finally, the meta-type *InfomationTerm* represents some primitive data type (String, Integer, Real, Boolean) or object type.

In addition to the generic fluents introduced in Section 3, our pattern language uses the following predefined fluents,

(i) *authorised(authorisingAgent,authorisedAgent,e)* – This fluent denotes that the agent *authorisedAgent* has been authorised to receive and process the event *e* or to send an event *e* by the agent *authorisingAgent*.

(ii) *exposes(o, owner, i)* – This fluent denotes that the response generated from the execution of an operation *o* will disclose an information term *i* which belongs to the agent *owner*.

(iii) *transforms(o, agent)* – This fluent denotes that the execution of the operation *o* may transform the state of the agent *agent*.

4.2 Pattern for Confidentiality

Confidentiality is defined in [3] as "absence of unauthorised disclosure of information". This property means that an agent who possesses some information should not allow unauthorised disclosure of it. Disclosure may occur either through direct access of the stored information at the side of the agent which possesses it or through the dispatch of the information. Information dispatch can happen through directed communications of the form *source* → *dest* in which the agent *source* who possesses the information sends it over to *dest*. In communication chains of the form *source* → $dest_1$ → … → $dest_n$ there is the possibility of unauthorised disclosure of information not only at the source or during the transmission of information over a channel but also at some destination agent $dest_i$ which despite being authorised to receive the information itself fails after receiving it to prevent unauthorised access to it by a third party. The term "unauthorised" in the above definition assumes an authentication and an authorisation process. Consequently, the knowledge of the authentication and authorisation status of agents is a necessary, albeit not sufficient condition, for monitoring confidentiality.

4.2.1 Pattern for source-party monitoring

The pattern for source-party confidentiality monitoring is shown in Figure 2.

The rule *CSR1* in this pattern is monitored at the source of an information disclosure message (i.e. the agent *_sender* in the pattern) and states that if an information disclosure event *E1* happens in this agent that allows another agent (*_receiver*) to obtain information *_i* which is confidential to an agent *_owner*, the *_receiver* should be authorised by the *_owner* to obtain *_i* at the time of the disclosure.

In the pattern, the authorisation of the *_receiver* is performed by an operation *_authorO* which tests whether the *_receiver* is authorised to receive the event *E1* and thus obtain the information *_i* of *_owner* at time *t*. During monitoring, authorisation can be obtained by deduction from the assumption *CSA2* of the pattern and the axioms of EC. According to *CSA2*, the fluent *authorised (<_owner>, <_receiver>, E1)* which indicates the authorisation of *_receiver* to receive the event *E1* that exposes *_i* is initiated only if an event *E2* that indicates the receipt of a response from the execution of the operation *_authorO* has occurred and the result of this operation (*_result_authorO*) signifies the authorisation (i.e., it is equal to *_authorisedValue*). Following the initialisation of the authorisation fluent *authorised(<_owner>,<_receiver>, E1)* at some time t_0 the predicate *HoldsAt(authorised(<_owner>, <_receiver>, E1), t1)* can be shown to hold at any time t1 after t_0 by the following axiom of EC [24] (assuming that no other event that could have clipped *authorised(<_owner>,<_receiver>, E1)* occurred between t_0 and t1):

$$\mathbf{HoldsAt}(f,t_B) \Leftarrow (\exists e,t)\ \mathbf{Happens}(e,t,\Re(t_A,t_B)) \wedge \mathbf{Initiates}(e,f,t) \wedge \neg\mathbf{Clipped}(t,f,t_B)$$

Monitoring rules:	
∀ _o:Operation; _i:InformationTerm; t1 :Time _sender, _receiver, _owner, _agent1:Agent; **Happens**(***E1***, t1, ℜ(t1, t1)) ∧ **HoldsAt**(exposes(<_o>, <_owner>, <_i>), t1) ⇒ **HoldsAt**(authorised(<_owner>, <_receiver>, ***E1***),t1) where: ***E1*** = e(_eID1[1],<_sender>,<_receiver>, [**REQ-*|RES-***], <_o>, <_sender>)	CSR1
Assumptions:	
Initially(exposes(<_o>, <_owner>, <_i>))	CSA1
∀ _authorO: Operation; _sender, _receiver,_agent1, _owner:Agent; _authorisedValue, _result_authorO: Term; t:Time; **Happens**(***E2***, t,ℜ(t,t)) ∧ **HoldsAt**(equalTo(<_result_authorO>, <_authorisedValue>),t) ⇒ **Initiates**(***E2***, authorised(<_owner>,<_receiver>,***E1***),t) where: ***E2*** = e(_eID2,<_agent1>,<_sender>, **RES-A**, <_authorO>, <_sender>)	CSA2

Figure 2. Pattern for confidentiality monitoring

As indicated by the event *E2*, the operation *_authorO* in the pattern is executed in some agent *_agent1* following a request by the *_sender*. Thus, *CSR1* is satisfied only if prior to its checking, an event *E2* has occurred and by virtue of *CSA2* the authorisation fluent *authorised(<_owner>, <_receiver>, E1)* has been initiated.

The effect of events on the disclosure of information in the pattern is specified by the assumptions *CSA1* that specify which operations can expose confidential information.

Note that the event *E1* in the pattern can be either a request for the execution of an information disclosure operation *<_o>* (i.e., *REQ-** events) or a response generated by executing *<_o>* in the sender following an invocation sent earlier by the receiver (i.e., *RES-** events). Also, *CSR1* can refer to requests and responses that are to be dispatched (i.e., when the event status is *REQ-B* and *RES-B*, respectively) or that have been already dispatched (i.e., when the event status is *REQ-A* and *RES-A*, respectively).

[1] The variables indicating the identifiers of events in all the formulas in the paper are assumed to be of type *String* and universally quantified.

In cases where the status of *E1* is *REQ-B* or *RES-B*, *CSR1* provides scope for pre-emptive control as the relevant requests or responses can be blocked if the rule is not satisfied. In cases where the status of *E1* is *REQ-A* or *RES-A*, the rule provides scope only for reactive control as the relevant dispatches would have occurred when the rule is checked.

The pattern in Figure 2 covers also destination-party monitoring since *CSR1* can monitor invocations of an operation in an agent by observing the responses (*RES-**) to these invocations which are generated by the agent.

4.2.2 Pattern for multi-party monitoring

To ensure that a piece of confidential information *_i* which an agent (*agent2)* has received from another agent (*agent1*) will not be disclosed to an unauthorised third party (*agent3*), it is necessary to apply *CSR1* to *agent2*. This is possible, however, only if *agent2* agrees to provide information about the exchange of all the events between it and third parties that would disclose *_i*. In this case, to enable the monitoring of *agent2*, the variable *<_owner>* in *CSR1* should be *agent1*, *<_sender>* should be *agent2*, and *<_receiver>* should be *agent3*. With these variable settings, *CSR1* would check if *agent3* is authorized by *agent1* to receive an event from *agent2* that discloses *_i*. Furthermore, separate instances of *CSR1* and *CSA1* should be created for all the operations which *agent2* can invoke in a third party or execute after an invocation from a third party and would disclose *_i*.

4.2.3 Example

To illustrate the use of the confidentiality pattern, we use a case study based on an e-healthcare system supporting monitoring, assistance and provision of medication to patients with critical medical conditions that is described in [4]. In this case study, patients who have been discharged from hospitals with potentially threatening medical conditions can use an *e-health terminal* (EHT) − that is an e-health application installed on their PDAs − to contact an *emergency response centre* (ERC) for assistance and fast ordering of medication.

In one scenario of this case study, a patient who had suffered from a cardiac arrest, feels unwell and sends through his EHT a request for assistance to ERC. To establish the cause of the problem, ERC retrieves the patient's medical record through the EHT. From this record, ERC establishes that the patient's doctor is on vacation and broadcasts a message to a group of doctors known to be able to substitute the patient's doctor. A doctor D receives this message on his own EHT and replies immediately. ERC verifies D's ability to substitute for the patient's doctor for the specific assistance request. Following this, D's EHT interrogates ERC to receive the patient's medical data. D analyses all these data, identifies the most appropriate treatment, and writes the electronic prescription on his/her EHT which subsequently sends the prescription to ERC which forwards it to the patient's EHT after registering it. In this scenario, Campadello et al. [4] have identified the following confidentiality requirement:

"*A patient's substitute doctor can access the patient's medical data if and only if he is the selected doctor*" (i.e., *Req. 2.2.1.7* in [4])

Assuming the following operations of ERC,

(a) *fetchPatientData(docID:String,request:String,patInfo:MedicalRecord)* − This operation retrieves the medical record of a patient (*patInfo*) given (as input) a medical assistance request associated with the patient (*request*) and the identifier of a requesting doctor (*docID*)

(b) *verifyDoctor(docID:String, request:String, verified:Boolean)* – This operation verifies if a doctor (*docID*) can deal with a given request (*request*)

the above requirement can be monitored by an instantiation of the confidentiality pattern in Figure 2 covering the interaction between ERC and the EHT of doctor D.

Table 1. Mapping of variables of confidentiality pattern

Pattern Term/Variable	System Operation or Parameter
<_sender>	`ercID`
<_receiver>	`docEhtID`
<_agent1>	`ercID`
<_owner>	`ercID`
<_o>	`fetchPatientData`
<_i>	`patInfo`
<_authorO>	`verifyDoctor`
<_result_authorO>	`verifyDoctor:verified`
<_authorisedValue>	`True`

To instantiate the pattern and specify the specific confidentiality properties of the system, system providers need first to identify the different agents and interactions in the system, and subsequently establish the sender and receiver for each interaction as well as the agents that can accept an event catcher. In the above example, we focus on the interaction from the doctor's EHT to the ERC and assume that an event catcher can be inserted at ERC's side. Note that here we consider ERC to be the sender since we are interested in ERC's reply to the doctor's EHT request for a patient's record.

Having identified the specific pattern rule that will be used, system providers need to identify the system variables/operations which will substitute for the pattern terms. To do so, they must define a mapping from pattern variables onto system operations and variables. An example of this mapping for the variables of the pattern of Figure 2 is shown in Table 1. Once this mapping is identified, system providers need to indicate the variables of the formula that will substitute for the parameters of the system operations. If such variables are not identified, formula variables with the same names and types as operation parameters will be automatically generated to substitute for these parameters. In our example, as Table 1 does not define a mapping of operation parameters, such default variables will be generated automatically for these parameters. Thus, given the mapping of Table 1, the following rule and assumptions can be generated automatically from the pattern of Figure 2:

```
Rule CR1:
∀ _eID1,_ercID,_docEhtID,_request:String;
  _patInfo: MedicalRecord; t1:Time
  Happens( e(_eID1,_ercID,_docEhtID, RES-B,
  fetchPatientData(_docID,_request,_patInfo),
  _ercID),t1,ℜ(t1,t1)) ∧
  HoldsAt(exposes(fetchPatientData(_docID,_request
  , _patInfo), _patInfo), t1) ⇒
  HoldsAt(authorised(_ercID,_docEhtID,
  e(_eID1, _ercID,_docEhtID, RES-B,
  fetchPatientData(_docID,_request,_patInfo),
  _ercID)), t1)

Assumption CA1:
  Initially(exposes(fetchPatientData(_docEhtID,_re
  quest,_patInfo),_patInfo))

Assumption CA2:
∀ _eID1, _eID2,_ercID,_docEhtID:String;
  _verified: Boolean; t:Time
  Happens(e(_eID2,_ercID,_ercID, RES-A,
  verifyDoctor(_docID,_request,_verified),
  _ercID), t,ℜ(t,t)) ∧
  HoldsAt(equalTo(_verified, True),t) ⇒
  Initiates(e(_eID2,_ercID,_ercID, RES-A,
  verifyDoctor(_docID,_request,_verified),
  _ercID), authorised(_ercID,_docEhtID,
  e(_eID1, _ercID,_docEhtID, RES-B,
  fetchPatientData(_docID,_request,_patInfo),
  _ercID)), t)
```

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

Note also that since the mapping of Table 1 does not define a mapping for the parameters *docID* of the operations *fetchPatientData* and *verifyDoctor*, a default rule variable called *_docID* has been generated for both these operations in the formulas and, as a result, the doctor's ID (*_docID*) to be used in the *verifyDoctor* operation will be the same as the one used in the *fetchPatientData* operation.

4.3 Pattern for Integrity

Integrity has been defined in [3] as "absence of unauthorised system state transformations". This definition implies that: (a) no unauthenticated and unauthorised agent should be allowed to request the execution of an operation that would change the state of another agent. Such unauthorised changes can be checked by destination-party monitoring as we assume that changes of system states may occur only at specific local agents and an external agent can only change the state of a system by calling operations at a destination agent.

4.3.1 Pattern for destination-party integrity monitoring

The pattern for destination-party integrity monitoring is specified in Figure 3. The monitoring rule of this pattern (*IDR1*) specifies that upon the receipt of a request from a *_sender* for the execution of an operation *_o* which may transform the system state of a *_receiver*, the *_sender* must be authorised by the *_receiver* to execute *_o*. The possibility of the execution of *_o* causing a transformation in the state of the *_receiver* is indicated by the fluent *transforms(E1, <_receiver>)* which the rule requires to hold when the *_receiver* gets the request.

Note that the pattern for integrity monitoring is similar to the one for confidentiality monitoring but differ in two points. The first difference is that the confidentiality pattern is monitored at the sender side, since it is the sender which may expose some information. The integrity pattern, on the other hand, needs to be monitored at the receiver side, since it is the receiver which may eventually transform the system state (at the request of the sender). This is shown by the different source terms of the events of the two patterns. The second difference between these two patterns is that the confidentiality pattern uses the *exposes* fluent to specify information disclosure while the integrity pattern uses the *transforms* fluent to specify system state transformations.

Monitoring rules:	
∀ _o:Operation; _sender, _receiver: Agent; t:Time **Happens**(***E1***, t, $\Re$(t, t)) ∧ **HoldsAt**(transforms(<_o>, <_receiver>), t) ⇒ **HoldsAt**(authorised(<_receiver>, <_sender>, ***E1***), t) where: ***E1*** = e(_eID1,<_sender>,<_receiver>,**REQ-***, <_o>, <_receiver>)	IDR1
Assumptions:	
Initially(transforms(<_o>, <_receiver>))	IDA1
∀ _authorO: Operation; agent1: Agent; t:Time _sender,_receiver, _result_authorO: Term; **Happens**(***E2***, t,$\Re$(t,t)) ∧ **HoldsAt**(equalTo(<_result_authorO>, <_authorisedValue>),t) ⇒ **Initiates**(***E2***, authorised(<_receiver>, <_sender>, ***E1***), t) where: ***E2*** = e(_eID2,<_agent1>,<_receiver>, **RES-A**, <_authorO>, <_receiver>)	IDA2

Figure 3. Pattern for destination-party integrity monitoring

4.3.2 Example

In the scenario outlined in Section 4.2.3, the following integrity requirement has been identified:

"*Electronic prescriptions shall be issued only by doctors by means of an e-health terminal.*" (i.e., *Req. 2.2.1.15* in [4])

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

(i) ERC provides the operation *createPrescription(docID:String, request:String, presc: Prescription)*to create new electronic prescriptions (*presc*) for a medical assistance request (*request*), and
(ii) doctors are authorised through the execution of the operation *verifyDoctor* of ERC as discussed in Section 4.2.3.

Table 2. Mapping of variables of integrity pattern

Pattern Term/Variable	**System Operation or Parameter**
<_sender>	`docEhtID`
<_receiver>	`ercID`
<_o>	`createPrescription`
<_authorO>	`verifyDoctor`
<_result_authorO>	`verifyDoctor:verified`
<_authorisedValue>	`True`
<_agent1>	`ercID`

Following the procedure for instantiating a pattern that we described in Section 4.2.3, we can define the mapping between the variables of the pattern of Figure 3, and the operations of the e-healthcare system and their parameters. This mapping is shown in Table 2. From this mapping we can generate the following rule as an instance of the rule *IDR1* in order to check the requirement *Req. 2.2.1.15* at runtime:

```
Rule IR1:
∀  _eID1,_ercID,_docEhtID:String; t:Time
  Happens( e(_eID1,_docEhtID,_ercID,REQ-B,
  createPrescription(_docID,_request,_presc),
  _ercID), t, ℜ(t,t))  ∧ HoldsAt(transforms(
  createPrescription(_docID,_ request,_presc),
  _ercID), t) ⇒ HoldsAt(authorised(_ercID,
  _docID, e(_eID1,_docEhtID, _ercID,REQ-B,
  createPrescription(_docID,_ request,_presc),
  _ercID)), t)
```

We can also generate the assumptions *IA1* and *IA2* below as instances of the formulas *IDA1* and *IDA2*, respectively:

```
Assumption IA1:
  Initially(transforms(createPrescription(_docID,
  _ request,_presc), _ercID))
```

```
Assumption IA2:
∀ _eID2,_ercID,_docEhtID:String; t:Time;
  _request: String, _verified: Boolean
  Happens(e(_eID2,_ercID,_ercID, RES-A,
  verifyDoctor(_docID, _request,_verified),
  _ercID), t,ℜ(t,t)) ∧
  HoldsAt(equalTo(_verified, True),t) ⇒
  Initiates( e(_eID2,_ercID,_ercID, RES-A,
  verifyDoctor(_docID,_request,_verified),
  _ercID), authorised(_ercID, _docEhtID,
  e(_eID1,_docEhtID, _ercID,REQ-B,
  createPrescription(_docID,_ request,_presc),
  _ercID)), t)
```

The rule *IR1* above checks whether the doctor (*_docID*) who invokes the operation *createPrescription* in ERC (*_ercID*) is authorised to do so. Note that, due to the mapping of the pattern variables of Table 2, *IR1* effectively describes a delegation of the doctor's right to create prescriptions to his/her EHT, since it is the EHT which is the sender in this interaction (*_docEhtID*), while it is the doctor who is being authorised (*_docID*) for the action in reality. The assumption *IA2* above states that a doctor is authorised to call the operation *createPrescription* in ERC only if this is verified by the operation *verifyDoctor*. In this case, an appropriate authorisation fluent will be generated by *IA2* and by virtue of the EC axiom discussed in Section 4.3.2 we can derive that the *HoldsAt* predicate in the head of the rule *IR1* is satisfied.

4.4 Pattern for Availability

According to [3]**,** availability is defined as "readiness for correct system service". In [3], a service is deemed to be correct if it implements the specified system function. Readiness of a system in this definition means that if some agent invokes an operation to access some information or use a resource, it will eventually receive a correct response to the request. In some cases, the property may be strengthened to require that a response will be received within a fixed time period following the invocation. These cases can be effectively monitored by our framework as indicated below.

4.4.1 Pattern for source-party availability monitoring

The pattern for source party availability monitoring specifies a monitoring rule that checks whether, following the dispatch of an event by a source agent (*_sender*) requesting the execution of an operation *_o* in a destination agent (*_receiver*), the source agent receives a response from the destination agent for the request within t_u time units after the dispatch (see rule *ASR1* in Figure 4).

Note that, as it is based on events captured at the source of a request, *ASR1* checks the availability of both the receiver of the request and the communication channel between it and the sender. Also the use of a bounded range for the time variable *t2* in *ASR1* (i.e., $R(t1,t1+t_u)$) is necessary since if the variable was unbounded, the rule would not be decidable (only cases where the rule is satisfied would be detectable).

Monitoring rules:	
∀ _o:Operation; _sender, _receiver:Agent; t1, t2:Time; **Happens**(e (_eID, <_sender>,<_receiver>, **REQ-***, <_o>,<_sender>), t1, R(t1,t1)) ⇒ **Happens**(e (_eID, <_receiver>, <_sender>, **RES-B**, <_o>,<_sender>), t2, R(t1, t1+ t_u))	ASR1

Figure 4. Pattern for source-party availability monitoring

4.4.2 Patterns for destination-party availability monitoring

For destination-party availability monitoring, we introduce two patterns that specify different monitoring rules. These patterns are shown in Figure 5. The rule of the first pattern (*ADR1*) states that an agent (*_receiver*) which receives a request for the execution of an operation from another agent (*_sender*) should respond to the sender within t_u time units after the receipt of the request. The rule of the second pattern (*ADR2*) is used to check the availability of an agent (*_sender*) which is known to operate correctly only if it requests the execution of an operation *_o* (or a set of such operations) in another agent (*_receiver*) at regular time intervals.

Pattern 1	
Monitoring rules:	
∀ _o:Operation, _sender, _receiver: Agent; t1, t2:Time **Happens**(e(_eID, <_sender>,<_receiver>,**REQ-B**, <_o>,<_receiver>), t1, R(t1,t1)) ⇒ **Happens**(e(_eID, <_receiver>, <_sender>,**RES-A**, <_o>,<_receiver>), t2, R(t1,t1+t_u))	ADR1
Pattern 2	
Monitoring rules:	
∀ _o:Operation, _sender, _receiver: Agent; t1, t2:Time **Happens**(e(_eID1, <_sender>,<_receiver>,**REQ-***, <_o>,<_receiver>), t1, R(t1,t1)) ⇒ **Happens**(e(_eID2, <_sender>,<_receiver>,**REQ-***, <_o>,<_receiver>), t2, R(t1,t1+t_u))	ADR2

Figure 5. Patterns for destination-party availability monitoring

Note that, in the case of availability there is no need for an additional multi-party monitoring pattern since if both source and destination party monitoring can be applied, the monitoring of *ADR1* checks the availability of the service at the destination side and the monitoring of rule *ASR1* checks the availability of both the destination side and the communication channel between the two agents. It should also be noted that the patterns for availability do not address the correctness of system functions. This is because system correctness must be assessed against some additional model of the intended behaviour of a system which can not be specified in a generic form as part of the pattern.

4.4.3 Example

In the e-healthcare system scenario introduced in Section 4.2.3, the following availability requirement has been identified:

> "*The patient's e-health terminal shall continuously… emit an OK status to the ERC.*" (i.e., *Req. 2.2.1.11* in [4])

Assuming that the status of an EHT to ERC is notified by calling the operation *reportStatus(ehtID: String)* and that the abstract *_sender* and *_receiver* terms in the pattern are mapped onto the id of an EHT (*_ehtID*) and the id of the ERC (*_ercID*) respectively, the above requirement can be monitored using the following rule

```
AR3: ∀ _eID1, _eID2, _ehtID, _ercID:String; t1,
      t2:Time Happens(e(_eID1,_ehtID,_ercID,REQ-A,
      reportStatus(_ehtID)),_ercID),t1, ℜ(t1,t1))
      ⇒  Happens( e(_eID2,_ehtID,_ercID,REQ-A,
      reportStatus(_ehtID),_ercID),t2, ℜ(t1,t1+tu)
```

AR3 is created by instantiating the rule *ADR2* of the availability pattern and checks if an EHT invokes the operation *reportStatus* in intervals of no more than t_u time units.

5. RELATED WORK

Konrad and Cheng [17] define specification patterns expressed in the real-time temporal logics MTL [18], TCTL [2], and RTGIL [21]. Their work extends the Dwyer et al. pattern system [10] with a taxonomy of (i) *duration properties* that describe a bounded duration of an occurrence, (ii) *periodic properties* that describe periodic occurrences, and (iii) *real-time order properti*es that place time bounds on the order of occurrences. Our patterns differ in that we specifically consider monitorable security properties and not general liveness/safety properties as [17].

Propel [25] is also an extension of [10] that helps specifiers identify the subtleties and alternative options associated with the intended behaviour of systems (see the *Response* pattern of [10] for example). Propel allows specifiers examine all the options explicitly and decide on the final instantiation of the property (where all the options have been resolved). Unlike our approach, Propel does not support the expression of real-time information.

Bandera [7] is another extension of [10] that deals with events in a state-based formalism but with no real-time information. Bandera supports user defined patterns and provides a structured-English front end for the patterns, which are then translated into the formalism of the chosen model checker.

Security patterns have also been introduced to aid the development of secure software systems but not for verification or monitoring. Such patterns have been proposed as part of UMLsec [15] and SecureUML [19]. In UMLsec, Jürjens proposes some transformation patterns between UML models that are used to introduce patterns by refinement [14]. These patterns are for security solutions rather than for formulating security properties. SecureUML [19] is another extension of UML which provides syntax for expressing access control policies directly in UML. Other recent work has also focused on specifying security patterns [23] that describe solutions to particular recurring security problems. Security patterns that can be applied at different architectural levels of software systems are also described in [12]. These patterns however are not expressed in a formal language and thus are not always clear to developers.

Finally, apart from [8] which describes *authorisation* policies for determining access rights of processes to objects (without being able to express time constraints), no other work in the area of intrusion detection and dynamic monitoring of security (e.g. [11][8][16][22][9]) has used property specification patterns to express monitoring rules, to the best of our knowledge.

6. CONCLUSION AND FUTURE WORK

In this paper, we have presented patterns for expressing three basic security properties, namely confidentiality, integrity and availability, in a way which permits their monitoring at runtime. These patterns are expressed in Event Calculus and can be used with a generic framework for monitoring functional and non functional requirements at runtime with sophisticated reasoning capabilities [26]. The creation of these patterns aims at making it easier for system providers to specify basic security requirements for their systems and monitor them using this framework. By doing so, system providers can outsource security requirements to the monitoring infrastructure and use monitoring as an additional layer of security checking which is independent from checks performed by the system that is being monitored, thus, making it more fault tolerant. Furthermore, monitoring enables the identification, diagnosis and future prevention of attacks, even for dynamic and difficult to describe attack scenarios.

Our current work focuses on the development of a tool to support the instantiation of the patterns presented in this paper and the validation of the generated security requirements. This tool will generate pattern instances automatically based on mappings between the pattern variables and system operations/variables as discussed in this paper. Our aim is to use this tool to support the instantiation of patterns for real-world applications and investigate if our approach is expressive enough for these. Finally, we are also working on the development of template formulas as part of the confidentiality patterns to assist system providers express theories for confidential information leaks that can result from disclosing non confidential pieces of information independently.

ACKNOWLEDGEMENTS

The work reported in this paper has been funded by the European Commission under the Information Society Technologies Programme as part of the integrated project SERENITY (contract IST-027587).

REFERENCES

[1] Specification patterns, http://patterns.projects.cis.ksu.edu/

[2] Alur, R.: Techniques for Automatic Verification of Real-Time Systems. PhD thesis, Stanford Univ., August (1991)

[3] Avizienis A., Laprie J-C., Randell B.: Fundamental Concepts of Dependability. Report N01145, LAAS-CNRS, (2001)

[4] Campadello S. et al.: S&D Requirements specification, Deliverable A7.D2.1, SERENITY Project, /www.serenity-forum.org/-Activities-.html?debut_article=4, (2006).

[5] Chen, L. and Avizienis A.: *N-version Programming: A Fault-Tolerance Approach To Reliability Of Software Operation, Proc. Of 25th Inter. Symp. on Fault-Tolerant Computing, 'Highlights from Twenty-Five Years', vol. III:113-119, 1995.

[6] Clarke, E.M., Emerson, E.A., and Sistla, A.P.: Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. ACM Trans. on Programming Languages and Systems, 8(2):244-263 (1986)

[7] Corbett, C., Dwyer, M.B., Hatcliff, and Robby: A language framework for expressing checkable properties of dynamic software. Proc. of the SPIN Software Model Checking Workshop, LNCS vol. 1885, (2000)

[8] Damianou N, Dulay N, Lupu E, Sloman M. : The Ponder Policy Specification Language, POLICY 2001, (2001)

[9] Denning, D.: An Intrusion-Detection Model, IEEE Trans. on Software Engineering, 13(2):222-232. (1987).

[10] Dwyer, M.B., Avrunin, G.S. and Corbett, J.C.: Property Specification Patterns for Finite state Verification. Proc. Of 2nd Work. on Formal Methods in Software Practice, (1998)

[11] English, C., Terzis, S., Nixon, P.: Towards Self-Protecting Ubiquitous Systems Monitoring Trust-based Interactions, Proc. of UbiSys '04, (2004)

[12] Fernandez, E.B., and Pan, R.: A pattern language for security models. Technical report, Florida Atlantic University, published in PLoP (2001)

[13] Flake, S., and Mueller, W.: An OCL Extension for Real-Time Constraints. Advances in Object Modelling with the OCL, LNCS. Springer-Verlag, (2001)

[14] Jürjens, J.: Transformations for introducing patterns - a secure systems case study. In Work. on Transformations in UML, ETAPS 2001 Satellite Event, (2001)

[15] Jurjens, J.: UMLsec: Extending UML for secure systems development. Proc. of the 5th Int. Conf. on the Unified Modeling Language, LNCS, 2460: 412-425, (2002)

[16] Ko, C., Ruschitzka, M. & Levitt, K.: Execution monitoring of security-critical programs in distributed systems: A specification-based approach. Proc. of the IEEE Symposium on Security and Privacy, 175-187, (1997)

[17] Konrad, S. and Cheng, B. H: Real-time specification patterns. Proc of the 27th Int. Conf. on Soft. Engineering, 372-381 (2005)

[18] Koymans, R:. Specifying Real-Time Properties with Metric Temporal Logic. RealTime Systems, 2(4):255-299, 1990

[19] Lodderstedt, T., Basin, D. A., and Doser, J.: SecureUML: A UML-Based Modeling Language for Model-Driven Security. Proc. of the 5th Int. Conf. on the Unified Modeling Language LNCS, vol. 2460: 426-441, Springer-Verlag, (2002).

[20] Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer-Verlag (1992)

[21] Moser, L.E., Ramakrishna, Y.S., Kutty, G., Melliar-Smith, P.M., and Dillon, K.: A Graphical Environment for the Design of Concurrent Real-Time Systems, ACM Trans. on Software Engineering Methodology, 6: 31-79, (1997)

[22] Porras, P. A. and Neumann, P. G.: EMERALD: Event monitoring enabling responses to anomalous live disturbances, In Proc. 20th National Information Systems Security Conference, 353-365. (1997)

[23] Security Patterns, http://www.securitypatterns.org

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

[25] Smith, R.L., Avrunin, G.S., Clarke, L.A. and Osterweil, L.J.: Propel: An approach supporting property elucidation. In Proc. of the 24th Int. Conf. on Software Engineering, 11-21, May (2002)

[26] Spanoudakis, G. and Mahbub, K.: Non Intrusive Monitoring of Service Based Systems, Int. Journal of Cooperative Information Systems, 15(3): 325-358, (2006)

[27] Srivatsa, M. and Liu, L.: Securing Publish-Subscribe Overlay Services with EventGuard, Proc. of the 12th ACM Conf on Computer and Communications Security, 289-298, (2005)