A4.D2.2 – Basic set of Information Collection Mechanisms for Run-Time S&D Monitoring

C. Kloukinas, C. Ballas, D. Presenza, G. Spanoudakis

Document Number	A4.D2.2
Document Title	Basic set of Information Collection Mechanisms for Run-Time S&D Monitoring
Version	0.15
Status	Final
Work Package	WP 4.2
Deliverable Type	Report
Contractual Date of Delivery	31 August 2006
Actual Date of Delivery	11 October 2006
Responsible Unit	CUL
Contributors	CUL, ENG
Keyword List	S&D monitoring, runtime events, event capture
Dissemination level	PU

Change History

Version	Date	Status	Author (Unit)	Description
0.1	14/7/2006	Draft	Christos Kloukinas	Table of contents, indicative section contents
0.2	28/7/2006	Draft	Christos Kloukinas	First draft with event structure and description of event mechanisms (chapters/sections 1, 2, 3, 4, 5.3.2, 6, 7 and appendices A &B)
0.3	30/7/2006	Draft	Domenico Presenza	Event collection for tuple spaces
0.4	6/8/2006	Draft	George Spanoudakis	Integration of materials from ENG (sections 5.1, 5.2, 5.3.1), description of EC-Assertion
0.5	11/8/2006	Draft	Costas Ballas	Amendments to XML schema for events
0.6	21/08/2006	Draft	Domenico Presenza	Modified the text according to the comments to the previous version. Aligned Tuple Space Proxy Architecture to the General Architecture of figure 2. Completed specification of the Tuple Space Abstract Language. First draft of the mapping between the Tuple Space Abstract Language and the SERENITY event language. Added Appendix B.
0.7	25/8/2006	Draft	George Spanoudakis	Editing
0.14	25/9/2006	Draft	Costas Ballas	Merging of final ENG materials
0.15	11/10/2006	Final	Costas Ballas	Corrections for adherence to the quality requirements.

Executive Summary

This document specifies the basic set of information collection mechanisms that we have developed in the SERENITY project in order to support run-time security and dependability monitoring**.** The developed mechanisms include:

— Operating System event collection mechanisms, i.e. mechanisms that support the collection of events which relate to system calls to the operating platform on which an application runs, for example calls of operations for creating, reading and accessing files.

— Middleware layer event collection mechanisms, i.e. these mechanisms that support the collection of events which relate to any middleware on which an application may run including mechanisms for tuple space event collection, workflow engine event collection and communication Protocol event collection.

The development of the above mechanisms has been based on the definition of an XML event representation schema that we have specified in order to provide a uniform way of reporting the events captured by the above mechanisms to the monitoring components of the SERENITY framework. This schema is also defined in this deliverable.

Furthermore, the implementations of the above event collection mechanisms are provided as a source code archive that accompanies this deliverable. This archive includes also simple applications that can be used to deploy the developed mechanisms in order to demonstrate their use. The basic instructions for the installation and use of these applications have been given in this deliverable.

Additional mechanisms to support the collection of events directly from applications will be specified and implemented in the next deliverable on event collection mechanisms (A4.D2.4) that is due in month 15. As part of the investigation of possible mechanisms for event collection at the application layer we will experiment with:

— Aspect-based event collection

— Java bytecode instrumentation

— Source code instrumentation

Table of Contents

1. Introduction

This document presents the basic set of event collection mechanisms which we have developed in the SERENITY project in order to support run-time security and dependability monitoring**.**

In order to help the reader we will first give a short description of the S&D patterns in SERENITY so that it is easier to understand how the event collection mechanisms will be used in SERENITY. Then, we will briefly cover some general aspects of event collection and follow with the presentation of the architecture we consider, the structure of the information we collect and a description of the initial set of mechanisms we have developed for information collection.

1.1. S&D Patterns in SERENITY

In SERENITY, S&D requirements are described using what we call "S&D patterns". These patterns describe S&D requirements and generic solutions to them, i.e., a particular security protocol which can meet the specified requirement. Along with the solution, an S&D pattern contains a description of the context of the solution, i.e., the specific conditions under which this solution is applicable, as well as, a set of monitoring rules which must be monitored at run-time to ensure that the solution and therefore the system that incorporates it correctly meets its S&D requirements. The monitoring rules can be used either for asserting certain requirements which cannot be proven statically or for asserting the context conditions of each solution in each possible state of the running system which must be satisfied for the solution to work properly.

An S&D pattern may be implemented in different ways and for different systems by what has been termed in the project as "S&D pattern implementations". In general, an S&D pattern contains a single S&D requirement, a solution for this requirement, a set of conditions which must be satisfied for the solution to work, and a set of monitoring rules. It can also be associated with a number of different S&D pattern implementations.

The implementations of a pattern are responsible not only for implementing the solution itself but also for observing and emitting the events that the pattern demands to be monitored (by referring to them in its monitoring rules). As such, the implementations need to use what we call event collection mechanisms to catch the different events as they occur and notify them to the monitor.

1.2. Purpose of this deliverable

The objective of this deliverable is to specify and provide an implementation of a basic set of event collection mechanisms, which can be used by the designers/developers of the different implementations of SERENITY S&D patterns in order to ease the development of systems that want to deploy the SERENITY framework and become monitorable by it. The deliverable also specifies the form in which these events should be reported to the SERENITY framework and the mechanism for reporting them to it

1.3. Overview of the Monitoring Language

The monitoring engine that we envisage to use in SERENITY will use *EC-Assertion* as the language for expressing the monitoring rules which need to be checked for verifying requirements at run-time. A monitoring rule can express either a *functional* or a *quality* requirement and may be associated with *assumptions* specifying effects of the behaviour of a system (and its constituent components) that affect the satisfiability of the requirements expressed by the rules. At runtime the

monitoring engine checks whether the monitoring rules are satisfied. During this check, any assumptions that have been specified for the rules are also used to generate additional information about the effects of the behaviour of the system and its components which affect the satisfiability of the rules.

EC-Assertion has been defined as an extended form of *event calculus* (EC) which is a first-order temporal formal language thus allowing us to specify properties of dynamic systems which change over time.

1.3.1. Overview of Event Calculus

In EC properties of dynamic systems which change over time are specified in terms of *events* and *fluents*. An event 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. A fluent may, for example, signify that a specific system variable has a particular value at a specific instance of time. Fluents are initiated and terminated by events

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)*. The meaning of this predicate is that a fluent *f* starts to hold after the occurrence of an event *e* at time *t*. The termination of a fluent is signified by the EC predicate *Terminates(e,f,t).* The meaning of *Terminates(e,f,t)* 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 at time *t,* respectively.

1.3.2. Special types of fluents and events used in EC-Assertion

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 as follows:

relation(Object1, …, Objectn) (I)

where *relation* is the name of the relation that takes as arguments *n* objects (Object1, …, Objectn) that can be fluents or terms. A pre-defined relation for fluents that is commonly used is:

valueOf(variable, value_exp) (II)

The meaning of (II) is that the fluent signified by *variable* has the value *value_exp*. In (II),

— *variable* denotes a typed variable or a list of typed variables which may be:

- *System variables* – 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

- *Monitoring variables* – 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 the monitor of the system uses in order to reason about the system).

If *variable* has the same name as a variable in the monitored system then it denotes this variable and is treated as an internal *variable*. In all other cases, variable denotes a monitoring variable and its type is determined by the type of *value_exp* as described below.

— *value_exp* is a term that either represents an EC variable or signifies a call to an operation that returns an object of some type. The operation called by *value_exp* may be an internal operation that is provided by the monitoring framework or an operation that is provided by an external entity. If *value_exp* signifies a call to an operation, it can take one of the following two forms:

- *S:O(_Oid,_P1,…,_Pn)* that signifies the invocation of an operation O in an external component S.
- *self:O(_Oid,_P1,…,_Pn)* that signifies the invocation of the built-in operation O of the monitor.

In these forms,

- *_Oid* is a variable whose value identifies the exact instance of O's invocation within a monitoring session, and
- *_P1, …, _Pn* are variables that indicate the values of the input parameters of the operation O at the time of its invocation.

The internal operations which may be used in the specification of fluents are shown in Table 1. An example of an internal operation is *add(n1:Real, n2:Real):Real* that returns n1+n2.

In addition to the generic fluents introduced above, we are extending EC-Assertion with a set of predefined fluents to support the specification of security monitoring rules. Two such fluents are:

— *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*.

— *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*.

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. Events are described in EC according to the following generic form:

e(_id, _sender, _receiver, _status, _o, _source)

where:

— *_event* is the name of the event

— *_ID* is a unique identifier for the event

— *_sender* is the name of the entity that sent the message.

— *_receiver* is the name of the entity 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.

— *_o* is a list of arguments and their types that the operation/event takes.

— *_source* is the name of the agent that provided information about the event.

In addition to the EC predicates and event/fluent denoting terms that we overviewed above, formulas that express properties that can be monitored in EC-Assertion may use the predicates < and = to express time conditions (the predicate *t1* < *t2* is true if *t1* is a time instance that occurred before *t2*, and the predicate *t1* = *t2* is true if *t1* is a time instance that is equal to *t2*) and to compare values of different variables. Also an EC formula that expresses a monitorable property must specify boundaries for the time ranges ℜ(LB,UB) which appear in the *Happens* predicates – there are closed ranges, i.e., by saying that time instance *t1* is in ℜ(LB,UB) we mean that LB<=*t1 and t1*<=UB.

Operation	Description
add(n1:Real, n2:Real): Real	This operation returns n1+n2
sub(n1:Real, n2:Real): Real	This operation returns n1-n2
mul(n1:Real, n2:Real): Real	This operation returns n1* n2
div(n1:Real, n2:Real): Real	This operation returns n1/n2
append(a[]: list of <T>, e:T): list of <T> where T is Real, Int or String.	This operation appends e to a[].
del(a[]: list of <T>, e:T): list of <T> where T is Real, Int or String.	This operation deletes the first occurrence of e in a[].
delAll(a[]: list of <T>, e:T): list of <T> where T is Real, Int or String.	This operation deletes all occurrences of e in a[].
size(a[]: list of <T>): Int where T is Real, Int or String.	This operation returns the number of elements in a[].
max(a[]: list of <T>):<T> where T is Real, Int or String.	This operation returns the maximum value in a[].
min(a[]: list of <T>):<T> where T is Real, Int or String.	This operation returns the minimum value in a[].
sum(a[]: list of <T>):<T> where T is Real or Int.	This operation returns the sum of the values in a[].
avg(a[]: list of <T>): <T> where T is Real or Int.	This operation returns the average of the values in a[].
median(a[]: list of <T>):<T>	This operation returns the arithmetic median of the

where T is Real, Int or String.	values in a[].
mode(a[]: list of <T>): <T> where T is Real, Int or String.	This operation returns the most frequent element in a[].
new(type_name:String): ObjectIdentifier	This operation creates a new object instance of type T and returns an atom that is a unique object identifier for this object.

Table 1 - Built-in operations of EC-Assertion

If the variable *t* in such predicates is existentially quantified, at least one of LB and UB must be specified. These boundaries can be specified by using: (i) constant time indicators or (ii) arithmetic expressions of time variables *t'* which appear in *Happens* predicates of the same formula provided that the latter variables are universally quantified, and that the expression appears in their scope. If *t* is a universally quantified variable both LB and UB must be specified. *Happens* predicates with unrestricted universally quantified time variables take the form $Happens(e,t,\mathfrak{R}(t,t))$. These predicates express instantaneous events. Furthermore, a formula is valid in EC-Assertion if the time variables of all the predicates which include existentially quantified non-time variables, take values in time ranges with fixed boundaries. These restrictions guarantee the ability to check the satisfiability of formulas.

A monitoring specification in EC-Assertion is composed of:

— a *monitoring rule* which defines in a parameterised form the event calculus formulas that will need to be monitored at runtime, and

— 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 rule based on captured runtime events.

1.3.3. Example of EC-Assertion monitoring specifications

To illustrate the use of EC-Assertion in the specification of monitoring rules and assumptions, we use a case study based on an e-healthcare system supporting monitoring, assistance and provision of medication to patients with critical medical conditions based on smart-item technology that is described in the A7.D2.1 deliverable of SERENITY [6]. 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. [6] 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 [6])

Suppose that ERC provides the operation *fetchPatientData(docID:String, request:String, patInfo:MedicalRecord)* which 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*). Given the above operation, the requirement *Req. 2.2.1.7* can be monitored by a monitoring rule requiring that when a doctor's EHT invokes the operation *fetchPatientData* in ERC which will disclose confidential patient data, the doctor's ID that is provided as an input parameter to the operation *fetchPatientData* must be authorised to request the execution of this operation and therefore receive the relevant patient record. This rule is specified below:

Rule CR1:

```
∀ _eID1, _ercID, _docEhtID, _request:String; _patInfo: MedicalRecord;
t1,t2:Time
Happens(
e(_eID1,_ercID,_docEhtID, RES-B,
fetchPatientData(_docID,_request,_patInfo), _ercID),t1,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)
```

In this rule,

— the predicate

```
Happens(
e(_eID1,_ercID,_docEhtID,RES-B,
fetchPatientData(_docID,_request,_patInfo), _ercID),t1,ℜ(t1,t1))
```

denotes the occurrence of the event

```
e(_eID1,_ercID,_docEhtID,RES-B,
fetchPatientData(_docID,_request,_patInfo), _ercID)
```

that represents the response of the ERC to the invocation of the operation *fetchPatientData(_docID,_request,_patInfo)*

— the predicate

```
HoldsAt(exposes(fetchPatientData(_docID,_request,_patInfo),
_patInfo), t1)
```

denotes that the execution of the operation *fetchPatientData(_docID,_request,_patInfo)* will disclose patient data (*_patInfor*), and

— the predicate

```
HoldsAt(authorised(_ercID,_docEhtID,
e(_eID1, _ercID,_docEhtID, RES-B,
fetchPatientData(_docID,_request,_patInfo),_ercID)), t1)
```

denotes that a doctor's EHT is authorised by ERC to receive a response from the execution of the operation *fetchPatientData(_docID,_request,_patInfo)* will disclose patient data (*_patInfor*) (following an earlier invocation of this operation).

Assuming that the authorisation of a doctor's EHT to request the execution of the operation *fetchPatientData* for a specific patient is determined by the operation *verifyDoctor(docID:String, request:String, verified: Boolean)* of ERC which verifies if the doctor (*docID*) can deal with a given request (*request*), the monitoring of the above rule requires the specification of the following assumption:

Assumption CA1:

```
∀_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)
```

Whilst monitoring *CR1*, the assumption *CA1* is used to derive the authorisation of a doctor's EHT (*_docEhtID*) to receive a response from the execution of the operation *fetchPatientData* that will disclose the record of a specific patient (*_patInfo*). This information is derived by deduction from the execution of the operation *verifyDoctor*. More specifically, according to *CA1* the fluent *authorised(_ercID,_docEhtID,e(_eID1,_ercID,_docEhtID,RES-B,fetchPatientData(_docID,_request,_patInfo), _ercID))* which denotes the authorisation of a doctor's EHT (*_docEhtID*) to receive the results of the execution of the operation *fetchPatientData(_docID,_request,_patInfo)* that will expose *_patInfo* is initiated only if the event *e(_eID2,_ercID,_ercID, RES-A, verifyDoctor(_docID,_request,_verified), _ercID)* that indicates the dispatch of a response from the execution of the operation *verifyDoctor* has occurred and the result of this operation (i.e., the value of the variable *_verified*) is equal to a value that indicates the authorisation of the doctor that owns the EHT (i.e., *True*). Following the initialisation of the above authorisation fluent at some time t0 the predicate *HoldsAt(authorised(_ercID,_docEhtID, e(_eID1,*

_ercID,_docEhtID, RES-B, fetchPatientData(_docID,_request,_patInfo),_ercID)), t1) of the rule *CR1* can be shown to hold at any time t1 after t0 by the following axiom of EC [19][1]:

$$\mathbf{HoldsAt}(f,t2) \Leftarrow (\exists e,t)\ \mathbf{Happens}(e,t,\Re(t1,t2)) \wedge \mathbf{Initiates}(e,f,t) \wedge \neg\mathbf{Clipped}(t,f,t2)$$

Furthermore, the monitoring of *CR1* will require the initiation of a fluent representing the fact that the execution of the operation *fetchPatientData* will disclose confidential information about patients. This knowledge can be deduced from assumptions about information disclosure by operations. In our example, to indicate that the execution of the operation *fetchPatientData* will expose the data of a specific patient we can specify the following assumption:

Assumption CA2:

```
Initially(exposes(
fetchPatientData(_docEhtID,_request,_patInfo),_patInfo))
```

CA2 specifies that the operation *fetchPatientData* discloses *patInfo* and by virtue of the EC axiom

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

it can be used to deduce the predicate

`HoldsAt(exposes(fetchPatientData(_docID,_request,_patInfo),_patInfo),t1)`

in rule CR1.

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

---

[1] This is true assuming that no other event that could have terminated – i.e. clipped in terms of EC – the fluent *authorised(_ercID,_docEhtID, e(_eID1, _ercID, _docEhtID, RES-B, fetchPatientData(_docID,_request,_patInfo), _ercID))* occurred between t_0 and t1.

2. Basic Architecture for Event Capturing

In this section we will describe the basic architecture of the systems which may be monitored in SERENITY and their deployment architecture that provide the basis for designing and developing our event collection mechanisms.

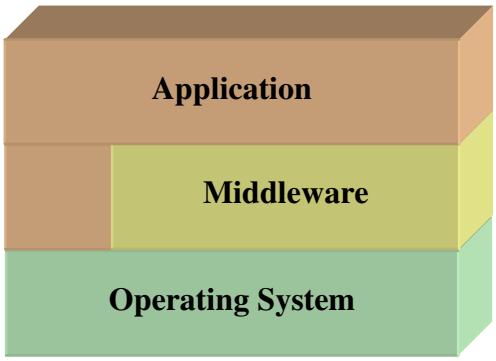


Figure 1 - Different layers in a system

As can be seen in Figure 1, we assume a general multi-teer deployment system architecture in which a software system is implemented and operates as a succession of different layers. Each of these layers uses the services offered by the lower layer(s). Typically, these layers include the Operating System (OS) layer, some middleware layer(s) above it (e.g., the Java virtual machine, workflow execution engines), and finally the application itself. In SERENITY, we are interested in observing events which occur both at the interfaces of the different layers, as well as, at the interior of these layers in certain cases. For example, we may wish to observe the value of an internal application variable (application layer internal event) or the interactions of the application and the OS (application-OS interface event).

Based on the generic system architecture of Figure 1, we distinguish the following generic categories of the event collection mechanisms that will be developed in SERENITY:

1. *Operating System event collection mechanisms* – these mechanisms will support the collection of events which relate to the operating platform on which an application runs, for example calls of operations for creating, reading and accessing files.
   - System Call event collection
2. Middleware layer event collection – these mechanisms will support the collection of events which relate to any middleware on which an application may run and will include mechanisms for:
   - Tuple space event collection
   - Workflow engine event collection
   - Communication Protocol event collection
3. Application Call event collection – these mechanisms will support the collection of events which relate to the application itself as, for example, calls of operations in the API of an

application, responses generated by the application after the invocation of operations, updates of internal application variables etc. The collection of such events may be based on:

- Aspect-based event collection mechanisms and construction of generic application wrappers that can catch the calls to an application and responses generated by it.
- Java bytecode instrumentation
- Source code instrumentation

In this deliverable, we concentrate on the Operating System and Middleware event collection mechanisms, i.e., system call and communication protocol event collection mechanisms. The collection mechanisms related to the application layer will be specified and implemented in the next deliverable on event collection mechanisms (A4.D2.4) which is due in month 15.

Table 2 presents a summary of the events that can be captured by the SERENITY event capturing mechanisms. The table indicates the types of the captured events and the deliverable in which the mechanisms for capturing them will be specified (i.e., this deliverable or A4.D2.4). Event types are characterised by the layer at which an event occurs (i.e., OS, Middleware and Application) and whether the event is internal to the layer or external (i.e., an interface event between two layers).

	A4.D2.2				**A4.D2.4**		
Layer	**OS**	**Middleware**			**Application**		
	System Call Event Collection	Tuple Space Event Collection	Workflow Engines Event Collection	SOAP Messages Event Collection	Java Source code Event Collection	Java Bytecode Event Collection	C/C++ Binaries Event Collection
Interface Events	✓	✓	✓	✓	✓	✓	✓
Internal Events			✓		✓	✓	✓

Table 2 – Overview of SERENITY event capturing mechanisms

It should be noted that certain mechanisms can be applied to more than one layer. For example, the application layer event collection mechanisms could also be applied to the middleware layer. This is because the application layer event collection mechanisms are generic enough to be applied to a middleware as well. For example, if the middleware has been implemented in Java then we can use either the Java source code or the Java bytecode to collect events which are internal to the middleware.

These event collection mechanisms can be used by the designers/developers of pattern implementations to capture the events mentioned in the monitoring rules specified in the relevant patterns. Of course there will be cases where these mechanisms do not suffice, either because they are not fast enough/small[2] enough or because the events are not in the categories we are considering

[2] In an embedded system, the memory footprint of the overall application can be very constrained – indeed, memory is usually the most expensive component of embedded systems.

in this deliverable. For example, there may be events internal to an OS, such as device driver status, which cannot be captured by our generic mechanisms. For such specialised events, the developers of pattern implementations will need to develop their own event collection mechanisms. Such non generic event capturing mechanisms may have their own internal specification and implementation but should adhere to the requirements of reporting events to the SERENITY framework.

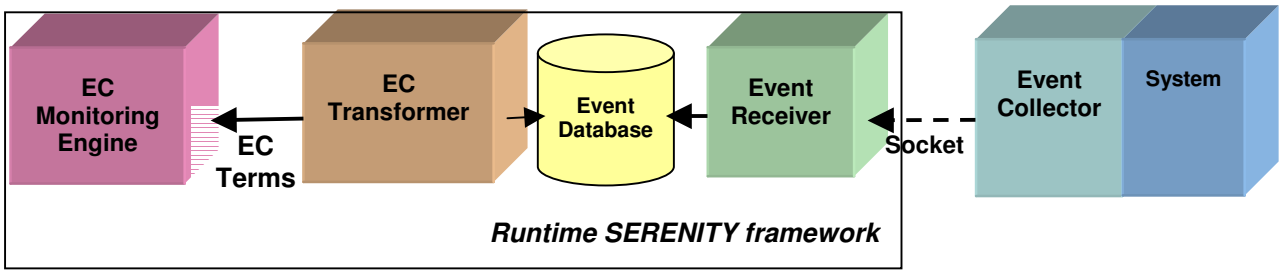


Figure 2 – Basic Architecture for Event Collection

These requirements have to do with the particular format that the collected events should have. To understand this requirement it is helpful to examine the generic architecture that the SERENITY will adopt for the collection of events and their reporting to its framework. This architecture is shown in Figure 2. As shown in this figure, first an *Event Collector* that is connected to a system that is being monitored or its deployment platform (e.g. middleware) captures the events of interest during runtime and transmits them through a socket to a remote *Event Receiver*. The transmitted events are represented in XML according to an XML schema that we define in Section 3. The Event Receiver is a component of the SERENITY framework that is responsible for storing the events that it receives in an event database. This database is accessed by another component of the SERENITY framework, called Event Calculus (EC) Transformer, at regular intervals. The EC Transformer retrieves all the new events that have been stored in the event database, translates them into Event Calculus terms and sends then to the EC monitoring engine. The EC Monitoring Engine accepts the reported EC terms and checks the monitoring rules against them.

The XML document which describes an event is being transferred as plain text. Many security issues rises as the data during this transportation can be attacked by a malicious user in order to prevent the normal functionality of the monitoring system or extract confidential information. Various kinds of attacks can take place during the transportation of the XML data such as modification of the data, exposure of data to unauthenticated users, or even attacks directly to the Event Receiver including, for instance, transmissions of forged XML documents in order to confuse the EC Monitoring Engine or cause denial of service. Thus, XML event documents must be protected. Mechanisms for the protection of the integrity, confidentiality, authenticity, accountability and authorization of XML documents have been introduced by the World Wide Web Consortium (W3C). XML Signatures [3], for instance can be used to authorize, provide non-repudiation mechanisms and check the integrity of XML documents, and XML Encryption [14] can provide document confidentiality. We are currently investigating these standards (along with their existing implementations) to establish if they can offer a sufficient level of security for the XML events required by our monitoring system and protect it from attacks. The adoption of such

mechanisms will be discussed in the second deliverable on event collection mechanisms (i.e., A4.D2.4)

3. Event Representation Schema

In this section we introduce the XML schema that we have developed to define the representation of events that are reported to the SERENITY framework. This schema is needed in order to provide a standard way of representing events which may be created by/observed at different layers of a system and by different event collectors (e.g., local function calls, remote procedure calls, SOAP messages, BPEL instructions, etc.).

Our schema assumes that interactions between the components of the systems which are being monitored are based on exchanging messages which may be calling functions or methods, or transmit data or signals. Generally, the description of a message should include:

— Message Type

  This field contains an unambiguous description of the message's type. A message can be either an operation or a communication of data (e.g. signals). When the message defines an operation this field contains the operation's name. That is, in the case of a polymorphic method, the name is a mangling of the operation name and the type of its arguments. In the case the message concerns a data communication then the data type and the data themselves are included in this field.

— Sender & Receiver

  These fields of an event identify the entity which initiated the event (sender) and the one which this event is directed to (receiver).

— Event Source

  This field describes the entity which the event was captured at (or emitted from), that is, the sender or the receiver.

— Event status (request / processing notification / response)

  This field identifies whether the event is a request for executing an operation to be executed (request), one which is currently being executed (processing notification) or one which has already been completed (response). The information herein will play a major role once we examine control mechanisms; indeed, it is much easier to control the execution of an operation which has not yet started, than one which is currently executing or has already finished.

— Event & Operation Correlation ID

  The event field contains an ID which can be used for correlating the current event with other events in the same "transaction"/"session". The operation ID appears only in the messages describing operations and it's used for correlating the current operation event with other events produced from the same operation. For example, it can be used to correlate an operation's response event to the respective operation's request event or a requested with a processed operation event.

— Collection Timestamp

  This field contains a timestamp which gives the time at which the event collector mechanism created this event; see Figure 2.

— Reception Timestamp

This field contains a timestamp which gives the time at which the event receiver at the monitoring engine side received this event, again see Figure 2.

— Data types and values

This field contains a portable description of the types of the parameters/results, as well as, the values of the input arguments.

Our need to support different system layers and implementation languages leads us towards a solution which is based on WSDL [7] descriptions of the data types.

— Context

The event context depends on the type of the event and the layer it is originating from. For example, in the case of system calls it can contain the time of the operation request/execution, the time that was reported to the monitor and the source of the event. Additional contextual information can be added in different implementations depending specific needs for monitoring.

Based on the above, we have defined an XML schema for describing events that represent messages. A graphical representation of this schema is shown in Figure 3 (continued in Figure 4) and its complete specification is given in 6. According to this schema, an event is described as an instance of the type *eventType* and is composed of:

— An element called *eventID* of type *unsignedLong*. This element is used to identify the event. Every event can be uniquely identified using this element with conjunction with the event source element.

— An element called *type* of type *EventType*. This element is used to provide information about the type of the event. An event can be: (i) a message related to the execution of an operation (i.e. a message that requests the execution of an operation or a message that notifies the completion of the execution of an operation) or (ii) a message that transmits data between systems (e.g. a signal). To capture both these cases, the complex type *EvenType* is composed of either an *OperationMessage* or a *DataMessage* element.

— An *OperationMessage* element is used in cases where a message calls or reports the results of an operation above and is of type *OpMsg*. The type *OpMsg* is composed of the elements *operationName, operationID, status* and *op_args* which are used to represent information related to the execution of an operation. More specifically,

- The element *operationName* is of type *string* and is used to specify the name of the operation.
- The element *operationID* is of type *long* and is used to identify the specific instance of an operation call or response within the execution of an application[3].
- The element *status* is of type *string* and is used to indicate the status of the operation execution event. The status of an operation execution event is: (i) REQ-B, if the message is a request for the execution of an operation that has been received but not processed yet,

[3] In cases where a message notifies the response from the execution of an operation, the value of *operationID* of the response message must be the same as the value of *operationID* of the message which called the operation.

(ii) REQ-A, if the message is a request for the execution of an operation that has been received and its processing has started, (iii) RES-B, if the message is a response ready to be dispatched but not dispatched yet, or (iv) RES-A, if the message is a response that has been dispatched.

- The element *op_args* is of type *argumentsType* and is used to specify the arguments of the relevant operation. The element *op*_args is optional because in some cases operations may have no arguments. The complex type *argumentsType* is a sequence of one or more elements of type *argumentType* that represent the arguments of the operation. The type *argumentType* defines a structure that is composed of either an element called *SimpleArgument* or an element called *Struct*. A *SimpleArgument* is used to represent operation arguments that have scalar values (e.g. real or integer values, strings etc). The type of this element is *simpleArgument* and is composed of the element's *name*, *type*, *value* and *argumentType* which represent the name, value and type of the relevant operation argument, respectively. The *argumentType* represents the type of the argument, i.e. if the argument is used as an input to the operation (then its value should be IN) or an output value from the operation (OUT). A *Struct* element is used to represent operation arguments of complex types. *Struct* is defined as a sequence of one or more *variable* elements of type *argumentType*. Thus, *Struct* can represent arguments of complex structures which may contain simple arguments as well as structured parts.

— A *DataMessage* element is used to describe messages that exchange data (e.g. signals). These elements are of type *DataMsg*. *DataMsg* has the elements *type* and *value* which are used to specify the type and the value of a datum exchanged by a data message.

— An element called *message_args* that is of type *MessageArgsList* and is used to provide information regarding the dispatch of a message. The type *MessageArgsList* is composed of three elements:

- The element *sender* which is of type *Entity* and indicates the entity that dispatched the relevant message. The type *Entity* is used to define any entity that can send or receive an event. Therefore it can be defined using any combination of the following elements:
  - *Name*: defining the name of the sender. In the case of BPEL for example, this may be the partner name which represents a web service we have invoked.
  - *ipAddress*: the IP address of the entity from which the event was produced.
  - *port*: Port number of the entity
  - *UserID*: the user ID that produced this event
  - *processID*: the process ID that produced this event or that the event refers to.
- The element *receiver* which describes the receiver of a message (i.e. the entity which will process the relevant message). The receiver is of the same type as the *sender* of the message, and therefore it can be defined using the complex element *Entity* as described above.
- The element *context* which is of type *Context* and is used to provide additional contextual information for the specific event. For now, this contextual information includes the time the event collector captured the event (see the element *collectionTime* in Figure 4), the time the event reported to the monitor (see the element *reportTime* in Figure 4), and the source of the event (see the element *eventSource* in Figure 4).

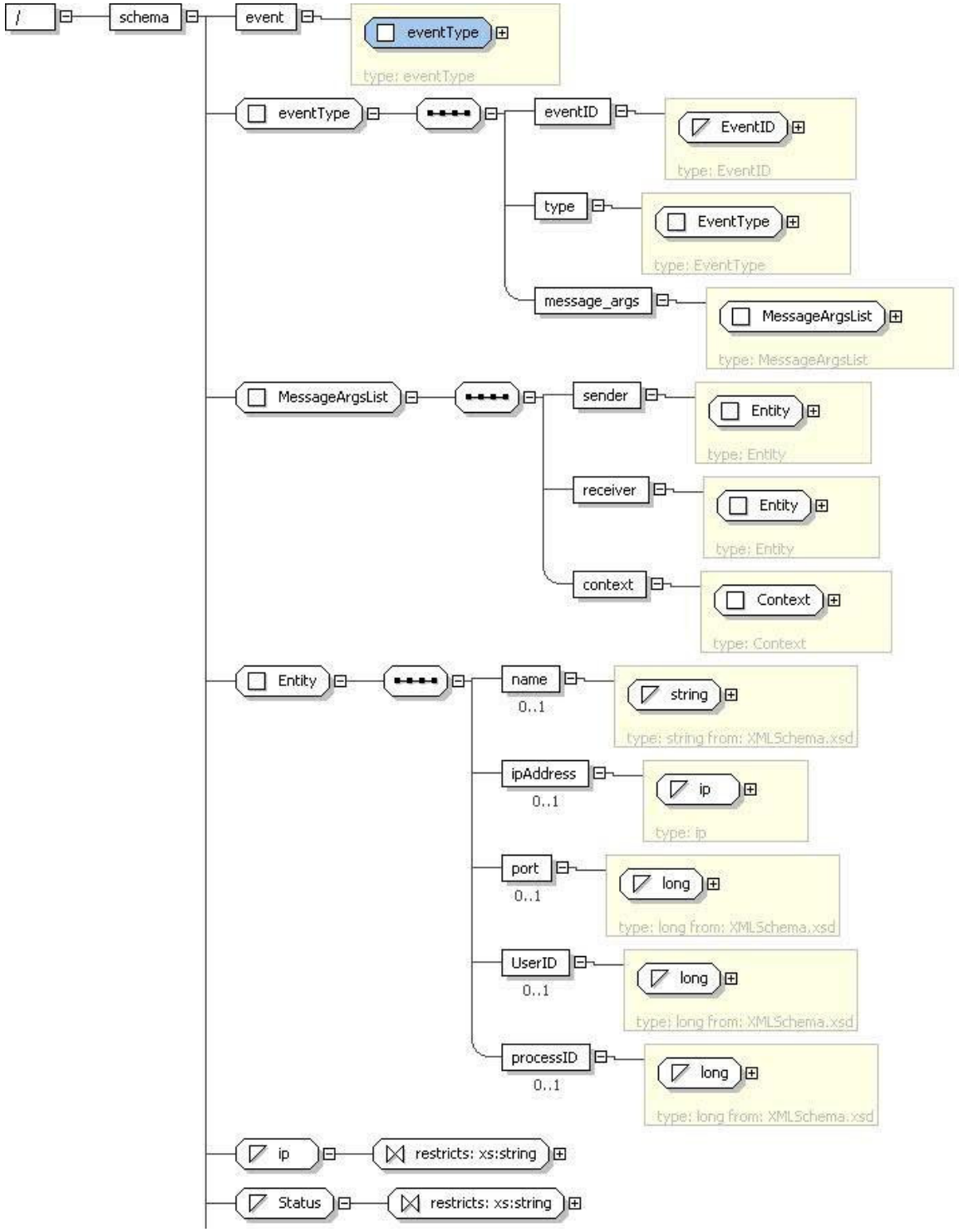


Figure 3 – XML Event Representation Schema (part I)

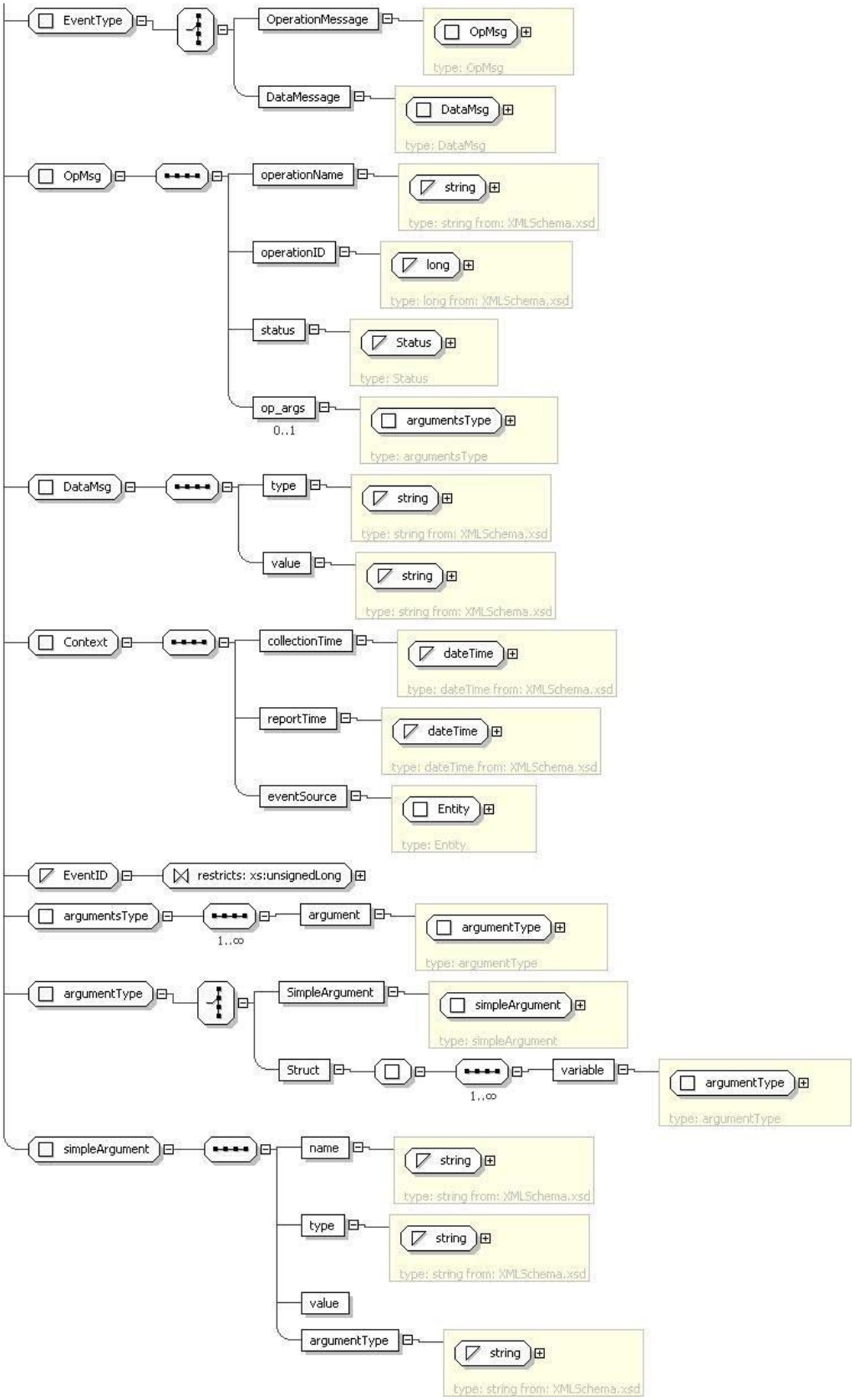


Figure 4 – XML Event Representation Schema (part II)

An example of an event described according to our event representation XML schema is given in Table 3. In this example of XML document we can see the representation of an event describing a system call from an application to the OS. The system call is for the operation open (line 07) and the arguments that it uses is the pathname for the file that is about to be accessed and the flags that represent what action is to be taken upon the file. The first argument is shown in lines 11 to 18, where we state the name, the type, the value and the type of the argument. The second argument is described in the same way between lines 19 and 26. The sender of this event is identified through the lines 31 to 35, where it is stated his IP address, his user ID and the process ID. Using the same way we represent the receiver of this event through the lines 36 to 40. In this case the receiver of the event is the OS, that's why the user ID and the process ID are set to zero. Any contextual information is described between the lines 41 to 49. For this event the contextual information is the time when the event was collected and the time in which it was reported and the source of the event. The combination of the fields which describe the event source (lines 44-48) and the event ID (line 4) can help us to identify uniquely this event.

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <event xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
03     xsi:noNamespaceSchemaLocation="file:/Z:/workspace1/events/events_v5.xsd">
04     <eventID>23</eventID>
05     <type>
06         <OperationMessage>
07             <operationName>open</operationName>
08             <operationID>2</operationID>
09             <status>REQ-B</status>
10             <op_args>
11                 <argument>
12                     <SimpleArgument>
13                         <name>pathname</name>
14                         <type>*char</type>
15                         <value>input.dat</value>
16                           <argumentType>IN</argumentType>
17                     </SimpleArgument>
18                 </argument>
19                 <argument>
20                     <SimpleArgument>
21                         <name>flags</name>
22                         <type>int</type>
23                         <value>0</value>
24                         <argumentType>IN</argumentType>
```

```
25                      </SimpleArgument>
26                  </argument>
27              </op_args>
28          </OperationMessage>
29      </type>
30      <message_args>
31          <sender>
32              <ipAddress>138.40.95.52</ipAddress>
33              <UserID>1002</UserID>
34              <processID>1617</processID>
35          </sender>
36          <receiver>
37              <ipAddress>138.40.95.52</ipAddress>
38              <UserID>0</UserID>
39              <processID>0</processID>
40          </receiver>
41          <context>
42              <collectionTime>2006-08-30T12:02:57</collectionTime>
43              <reportTime>2006-08-30T12:02:57</reportTime>
44              <eventSource>
45                  <ipAddress>138.40.95.52</ipAddress>
46                  <UserID>1002</UserID>
47                  <processID>1617</processID>
48              </eventSource>
49          </context>
50      </message_args>
51  </event>
```

Table 3 – Example of an XML document representing an event

4. Event Capturing at the OS layer

This is an event capturing mechanism that is positioned between the application/middleware and the operating system layer and is used to intercept system calls. By using this mechanism, the monitor is able to obtain the system calls along with their arguments and context (which it interprets as events). This information may be necessary for checking monitoring rules regarding the correct execution of an application.

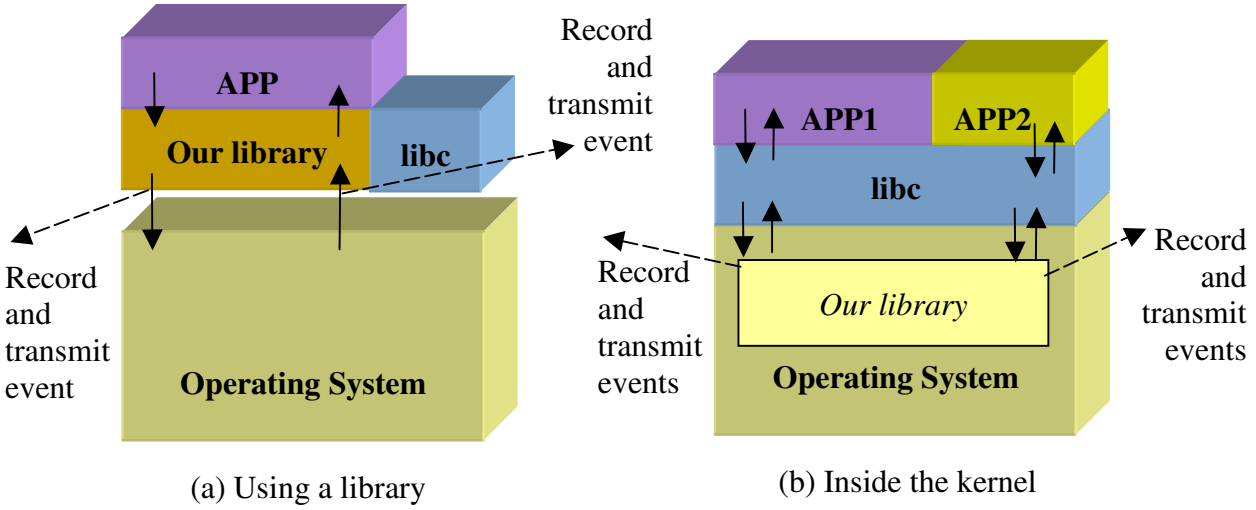
4.1. System calls proxies

For the OS layer, often it is necessary to be able to monitor the requests for system resources from an application (e.g. file access requests, process priority change requests etc). These requests occur in the form of so-called system calls. System calls form the API of the OS that is available to the application and are implemented as special procedures of a library provided along with the OS kernel that a user-space application can call to pass to the kernel the type of request it wants to make along with any relevant parameters. For example, in order to obtain access to a file, a user process needs to make a system call to the OS kernel and pass with it the name of the file, the way it wants to access the file (read, write, read-write), etc. Then the kernel checks the privileges of the process and the respective file, updates its internal data structures and returns either a file handle to the user application or an error indicating the reason for which the request could not be fulfilled.

The ability to monitor these requests is very important because it allows the monitor to verify the correct functionality of the system at the most basic layer, without which the higher level requests cannot operate correctly and securely. Our current implementation attaches the event collectors at the application when the latter starts executing. To understand how this is done we will first describe the use of system calls in a specific OS, namely Linux.

The OS makes available a number of different types of requests that a user process (UP) can make to it. The UP then uses the *syscall()* procedure to initiate these requests. For example, if UP wishes to find its group ID, it needs to call *syscall(SYS_getpgid, pid)*. Since the *syscall* procedure is a rather low-level and cumbersome API, there has been developed a higher-level API, which offers a single procedure for each type of system call request. This API is implemented inside the *libc* library. The *libc* library is the basic library that any Unix-like operating system uses to define its collection of system calls and other basic facilities. When applications are executed, they are first linked dynamically with *libc* so that they may make use of this higher-level API at runtime. Provided that it is dynamically linked to *libc*, the UP would need to call *getpgid(pid)* in order to find its group ID which is not only simpler but also allows the compiler to check the type of the argument.

In order to be able to observe the system calls we have developed a new library containing specialised implementations of the various system calls in *libc*. Each time a system call is made our implementation sends a message to a socket containing the description of the particular call and then forwards the call to the OS by using the low-level API of *syscall*. By placing our library at the front of the path for dynamically linked libraries we can ensure that the linker will chose our implementation of the system calls instead of the one found in *libc* (Figure 5(a)). Thus, system calls will be handled by the newly developed library, which can record and transmit the relevant information to the monitor, before delegating the responsibility for the execution of the system call to the OS kernel itself.



(a) Using a library

(b) Inside the kernel

Figure 5 – System call proxies

4.1.1. Advantages, Limitations & Extensions

The advantages of this approach are that it is relatively easy to develop this new library and that we do not need to change either the application code or the kernel of the underlying OS. The disadvantages on the other hand are the following. First, an application could have been linked statically with all the libraries it needs (including libc). In this case our library will be ignored and the application will communicate with the OS directly. Another possibility is for an application to not use the higher-level API of *libc* at all but instead use the low-level one of *syscall*. Again in this case our library will not be able to catch these system calls. These disadvantages will exist in any solution which wishes not to modify the OS kernel and are not specific to our solution.

The alternative that could be used if one wants to ensure that all system calls will be captured, independently of how the application has been developed and linked, would be to modify the kernel itself and introduce the code for capturing the system call events inside the kernel (Figure 5(b)). This is a possibility that we are planning to examine for the second deliverable on event collection mechanisms (i.e., A4.D2.4).

It should also be noted that we do not catch all the interactions between the OS and an application, independently of whether we use our current, library-based system call event collection implementation or a kernel-based one. The only interactions which are captured are the ones which occur through system calls. Other types of events that we do not catch are the asynchronous signals that the OS can send to an application (for example to kill it). These signals do not make use of the system call mechanism and as such are not covered by the mechanisms we have described in this section. To be more precise, what cannot be captured by the system call event collectors are the reception of a signal by an application and the invocation of a signal by the OS (to an application). We can, nevertheless, capture the invocation of a signal by an application towards some other application because this is performed using a system call (called kill).

Another aspect of the system call proxies which will be further investigated in the next deliverable is the optimisation of the event communication. Currently, a new socket connection is opened for each event, the event is transmitted and then the connection is closed. While this suffices for a proof-of-concept system, it is evident that there is a big room for improvement. A simple

optimisation we will consider is the reuse of the initial socket connection to send the future events as well, so as to avoid opening a new socket and reclaiming it for each event. Another one has to do with deferring the transmission of events, choosing to buffer them instead in a local-memory buffer. This buffer can then be read by another process periodically and that process will be the one consuming the events and transmitting them to the event receiver, thus allowing the application, the OS and the system call event-collector to not wait for the actual transmission of the events over the network.

4.1.2. Capture of System Calls

According to the description in the previous section the implementation of the collection of system calls is based on a library which is loaded with the application and acts as a middle layer between the application and the operating system. The additional functionality that this layer introduces is the transformation of the system calls to events, based on the XML schema described in section 3 and its transmission to the Event Receiver. Our implementation for the capturing of system calls is based on Debian GNU/Linux system (kernel ver. 2.4.27-2-386). In the implementation of the library we have implement a set of system calls demonstrating by this way how the development should be for the whole set of the system calls.

— Implementation

The implementation as mentions above is based on a GNU/Linux system. It has two components: the Event Receiver and the Event Collector library. The Event Receiver is a simple server application which can listen to a given port and display any incoming message. In this application we will be able to see the XML reports of events that the Event Collector library transmits. In order to execute the application we must go to the directory in which we untar the `SCProxies.tgz`, probably `$HOME/SCProxies/` and execute the command `./server address port`. The argument `address` specifies the IP address of the server and the `port` specifies the port number in which the application will listen for the events. For example `./server 138.40.95.52 9734` (Figure 6).

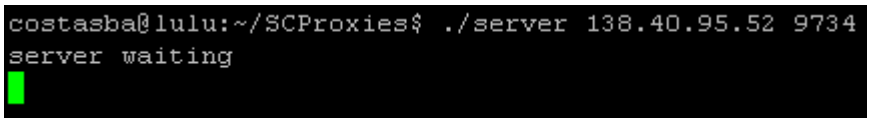


Figure 6 - Execution of Event Receiver

The event collection library is the `libfuncs.c`. There is the code for the system calls we can intercept. Before we try to compile the library we must edit it and modify the lines 75 and 76 specifying the IP address and port number of the event receiver. According to our example for the execution of the event receiver the lines shall be modified as shown in Figure 7.

```
address.sin_addr.s_addr = inet_addr("138.40.95.52");
address.sin_port = htons(atoi("9734"));
```

Figure 7 - IP/port modification

The "138.40.95.52" and "9734" are the IP address and the port number in which the Event Receiver is waiting for events.

After these modifications we are ready to compile the library giving the command `make` while we are in the same directory as `libfuncs.c`. For the demonstration of the library in the directory we included a testing application. This application doesn't do anything specific. It just calls a number of system calls that are modified inside our library in order to be give as the chance to see the creation of the events on the event receiver. To run this application using our library we must execute in the same directory the command `LD_PRELOAD=/home/costasba/SCProxies/libfuncs.so ./test_app` . The test application begins its execution informing us for the commands that it executes while we can observe in the event receiver the events generated from the execution of the application as shown in Figure 8.

```
server waiting
<?xml version="1.0" encoding="UTF-8"?>
<event xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="file:/Z:/workspace1/events/events_v4.xsd">
        <eventID>7</eventID>
        <type>
                <OperationMessage>
                        <operationName>getuid</operationName>
                        <operationID>1</operationID>
                        <status>RES-B</status>
                        <op_args>
                                <argument>
                                        <SimpleArgument>
                                                <name>uid_t</name>
                                                <type>uid_t</type>
                                                <value>1002</value>
                                                <argumentType>OUT</argumentType>
                                        </SimpleArgument>
                                </argument>
                        </op_args>
                </OperationMessage>
        </type>
        <message_args>
                <sender>
                        <ipAddress>138.40.95.52</ipAddress>
                        <UserID>0</UserID>
                        <processID>0</processID>
                </sender>
                <receiver>
                        <ipAddress>138.40.95.52</ipAddress>
                        <UserID>1002</UserID>
                        <processID>26300</processID>
                </receiver>
                <context>
                        <collectionTime>2006-09-13T12:04:12</collectionTime>
                        <reportTime>2006-09-13T12:04:12</reportTime>
                        <eventSource>
                                <ipAddress>138.40.95.52</ipAddress>
                                <UserID>0</UserID>
                                <processID>0</processID>
                        </eventSource>
                </context>
        </message_args>
</event>
```

Figure 8 - Reported Event

5. Event Capturing at the Middleware Layer

The systems targeted by SERENITY may comprise several different middleware layers. For example, a system may be using a virtual machine like the Java Virtual Machine (JVM), a Message Oriented Middleware (MOM) like a tuplespace, or a workflow engine. Typically, SERENITY patterns will specify monitoring rules defined in terms of events that are dispatched and received by each of these layers. Consequently, the SERENITY framework should provide a set of mechanisms for capturing such events. The mechanisms developed in the SERENITY project in order to capture events at the middleware layer are described in this section.

Our focus is the types of middleware targeted by the SERENITY framework, namely tuple spaces, workflow engines and communication protocols.

5.1. Tuple space proxies

Tuple spaces are shared repositories of data items which are aimed at supporting coordination and communication between concurrent processes. The tuplespace concepts were originally introduced by the Linda coordination language ([10]) but more recently the tuplespace model has been investigated also as a model to program coordination protocols in distributed computing settings ([9][18]). In tuple-based (data-driven) coordination models, client processes communicate and coordinate their activities by exchanging tuples of data via tuple spaces.

In this section we introduce the architecture proposed by SERENITY to collect messages exchanged between a tuplespace client and a tuplespace server.

The proposed architecture is shown in the Figure 5.

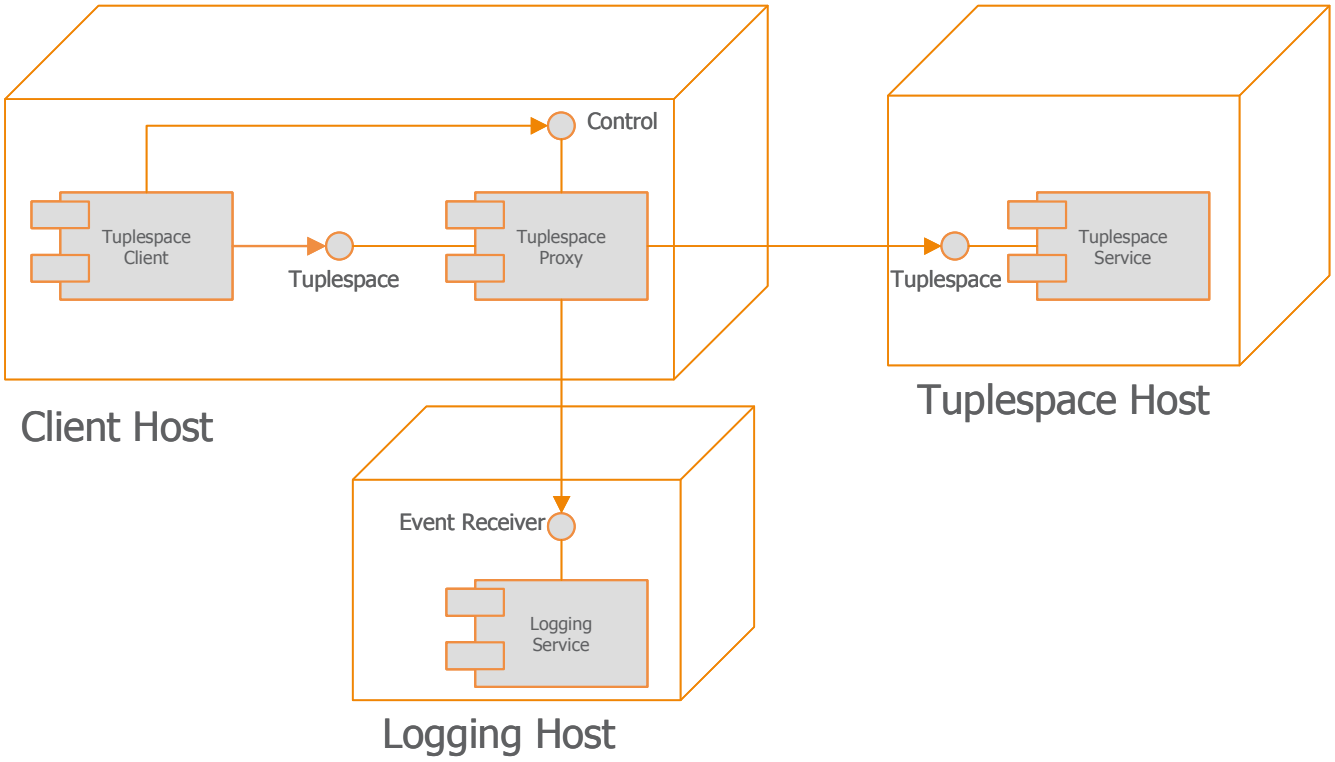


Figure 9 – Deployment of the SERENITY tuplespace logging subsystem

The components in this architecture are:

- *Tuplespace client*: This component is any application making use of a tuplespace service.
- *Tuplespace event collector*: This component intercepts tuplespace service requests. The component forwards tuplespace service requests to the remote tuplespace service and, at the same time, logs operations by means of the remote logging service.
- *Tuplespace service*: This component provides the tuplespace service. For testing our implementation we used the Outrigger tuplespace server available in the Jini environment [9].
- *Event Logging service*: This component provides the logging service via the Event Receiver interface (see also Figure 2 in Section 0).

5.1.1. The tuplespace event collector

A tuplespace event collector provides two interfaces to its clients:

- *Tuplespace interface*: This interface is used by the clients of a tuplespace proxy to submit service requests to a tuplespace service. This interface is specific to the actual tuplespace technology of the tuplespace service components. Our implementation is based on the JavaSpace specifications.
- *Configuration interface*: This interface is used by clients to control which tuplespace service requests operations have to be logged (e.g. clients running on nodes having connections with limited bandwidth might exploit this feature to reduce the network traffic). Developers of tuplespace proxies should implement this interface in order to render their proxies suitable for SERENITY;

A tuplespace proxy component also requires a *Logger* interface:

- *Event Receiver interface*: this interface is used by a tuplespace proxy component to log requests intended for a (possibly remote) tuplespace service.

5.1.2. A Tuplespace model for the Tuplespace interface

The Tuplespace interface is the interface that a Tuplespace Proxy offers to its clients to request operations on a tuplespace.

The lack of a standard interface for current tuple-based technologies (e.g. JavaSpace [9], TSpace [20], etc.) implies that the SERENITY Architecture cannot provide a single implementation of the Tuplespace proxy component and, hence, it is required to develop a proxy for each technology.

However, in the literature there are proposals for general frameworks able to capture the main ideas underlying the different tuple-based models subsumed by the different technologies. In particular the SERENITY Tuplespace logger follows the tuplespace framework proposed in [5].

In such a model each client is supposed to operate over a denumerable set of tuple spaces: each operation has the general form *op(…)@s* where s is a tuplespace service. In the description of the operations in the model, for the sake of clarity we omit the tuplespace service specification part.

The general model proposed by SERENITY for tuplespace defines the following operations to modify the content of a tuplespace:

— Basic operations defined in all Linda-like languages:

- *out(t)*: adds a tuple into a tuplespace;
- *in(p):* remove and return a tuple matching the template p from the tuplespace. The operation blocks until a tuple matching the p appears;
- *rd(p):* return (do not remove) a tuple matching the template p from the tuplespace. The operation blocks until a tuple matching p appears;

— Transaction operations. These operate on multiset of data:

- *rew(m1, m2):* atomically removes the tuples matching the template *m1* and then atomically produces the multiset data *m2*;

— Global operations. These operations require a global vision of the shared data space:

- *tfa(p):* verifies that no data matching a template *p* are available;
- *inp(p):* non-blocking version of the *take* operation;
- *rdp(p):* non-blocking version of the *read* operation;

— Global transaction operations. These are transaction operations which are able to test the global state of a shared data space:

- *collect(p)*: removes and returns all tuples matching the template *p*;
- *copy_collect(p)*: returns (does not remove) all tuples matching the template *p*;

Given the above model, a SERENITY proxy for Tuplespace will generate events according to the language described in Section 3.

The availability of a general framework enables the specification of the mapping of an expression from a technology specific tuplespace language (e.g. JavaSpace, Tspace, etc.) to the SERENITY event representation schema that we introduced in Section 3 in two steps:

1. from the technology specific language to the abstract tuplespace model;
2. from the abstract tuplespace model to the event representation language;

In this section we describe the first mapping that is valid for all technologies. Appendix B contains, as an example, the mapping between the JavaSpace model [9] and the general model.

The following table shows how the major components of the event representation language are used by SERENITY tuplespace proxies:

Event language element	Usage
operationName	Name of the tuplespace operation (e.g. *out*, *in*, *rd*).
status	REQ-B
op_args	One element for each field of the tuple given as actual parameter to the tuplespace

	operation. In our implementation, based on the JavaSpace specifications, an extra field is added to represent the type of the object given as actual parameter (this is required since the arguments of JavaSpace operations are objects and not tuples).
sender	TCP address of the host where the operation took place.
receiver	TCP address of the tuplespace services.
collectionTime	time the operation took place (expressed as host local time).

Table 4 – Usage of the Event language elements

5.1.3. Control Interface

The Control interface defines the following methods:

- *new(s)*: creates a new tuplespace service for the proxy;
- *moveTo(s)*: changes the tuplespace service referenced by the proxy
- *enableLogging(op)*: enables logging of operation op;
- *disableLogging(op)*: disables logging of operation op;
- *restoreDefaultLogging()*: restores the default logging settings for the proxy;

5.1.4. Logger Interface

The Logger Interface defines the following methods:

- *log(od)*: where *od* is a tuplespace operation description done with the event model described in paragraph 3.

5.1.5. Dynamic Behaviour

The following sequence diagram shows a typical interaction between the components of the Tuplespace logging subsystem:

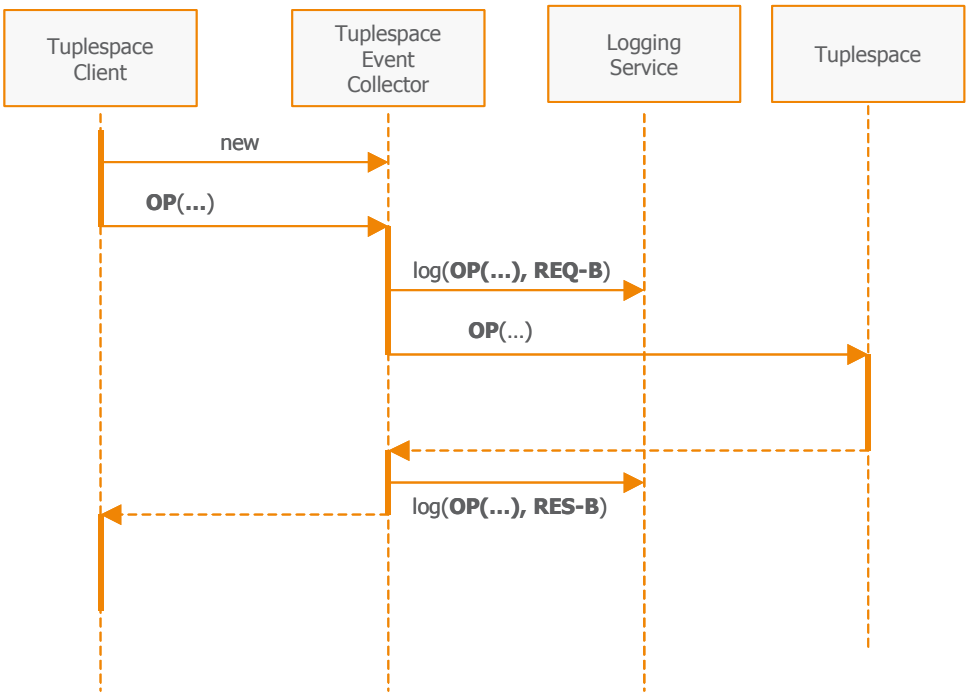


Figure 10 - Logging tuplespace operations

The interaction between components goes through the following sequence:

1. In order to access a Tuplespace a Tuplespace client creates a Tuplespace event collector. Upon initialisation a Tuplespace event collector discovers and binds a Tuplespace service.
2. Each time a Tuplespace event collector intercept an operation request for the Tuplespace service it logs the request to the Logging service;
3. The Tuplespace Event collector forward the operation to the Tuplespace service bound during step 1.
4. Upon reception of a response from a Tuplespace service the Tuplespace Event collector logs the event.

5.2. Workflow engine event capturing

The goal of this paragraph is to extend the set of event monitored by the SERENITY Architecture in the BPEL/Web-Services context and introduced in paragraph 1.2.2. Such an extension is needed to be able to deal with distributed ([15]) and/or QoS-aware ([22]) workflow engines.

5.2.1. Distributed engines events

Distribution of the orchestration process reduces inefficiencies introduced by a centralized control and improves performance and throughput. Distributed workflow engines typically aim to minimize communications costs and maximize the throughput of multiple concurrent instances of the input program.

In Ambient Intelligence (AmI) settings distribution of workflow engine has an additional motivation due to the fact that computational nodes have limited resources and none of them, in

principle, could be able to run a fully flanged workflow engine. In mobile information systems distribution of the workflow engine is the only option to orchestrate web services ([16]).

Typically in distributed workflow engines the execution of an instance of a process goes through the following steps:

1. The process definition is partitioned into a set of equivalent programs;
2. Each programs is assigned to a (possibly different) workflow engine;
3. All programs are executed in parallel by exploiting the computational resources available at the node that the programs have been assigned to.

The above general scheme may have some variation to provide the system with more adaptivity; for example the assignment of programs to nodes (step 2) might be redone during execution (step 3) to cope with variations in the availability of resources at nodes.

In SERENITY we introduce the following events to monitor the activity of distributed workflow engines:

— *acquire(hostID, engineID, procDefId, procId, activityName)*: the activity *activityName* belonging to the process *procDefId* has been assigned to the *engineID* workflow engine residing at *hostID* host. The assignment is valid for the execution of the *procId* instance of the process.

— *lose(hostID, engineID, procDefId, procId, activityName)*: the assignment signalised by an acquire event having the same actual parameters is no longer valid (It is going to be reassigned to a new node).

— *run(hostID, engineID, procDefId, procId, activityName)*: the activity *activityName* belonging to the process *procDefId* is being executed by the *engineID* workflow engine residing at *hostID* host. The run is part of the execution of the *procId* instance of the process.

5.2.2. QoS-aware engines events

QoS-aware workflow engines aim to maximize the Quality of Service (QoS) of composite service execution by taking into account the constraints and preferences of the users. QoS-aware engines rely on the concepts of abstract services and dynamic bindings. In this context composite services are described in term of abstract services that do not correspond to any specific implementation: loosely speaking abstract services are placeholders representing the whole set of actual “concrete” services sharing the same semantic. At design time the designer/developer of the composite service reason in term of abstract services.

At run-time, when the process instances are created and executed, the engines replace each abstract service (placeholder) with a concrete one (i.e. one running on a actual node on a network) selected from the set of semantically equivalent ones. The selection (binding) process takes into account both the QoS constraints defined on the process at design time and the constraints imposed by the user of the composite service.

Typically the selection process takes place whenever a new process instance is created. However, during the execution of a process instance some of these kinds of workflow engines have the possibility to measure the actual QoS of selected services ([22]). In case the actual measures indicate a possible violation of the QoS constraints the engine tries to recover by re-running the binding phase for the not yet executed service invocation activities.

Thanks to ability to measure actual QoS and to re-run the binding phase, QoS-aware workflow engines are able to react (adapt) to changes occurring during the execution of composite services.

The ability to react to changes is of extreme value in AmI settings where, due to the volatile nature of the involved networks, continuous and unpredictable variations of QoS are the norm.

Typically in a QoS-aware workflow engines execution of an instance of a process goes through the following steps:

1. *"concretisation" phase*: by consulting some sort of registry the engine binds each abstract service to a concrete one satisfying QoS constraints and trying to optimise some goal function;
2. *execution and monitoring phase*: the engine carries on the execution of the process instance by utilising the binding established in the previous phase. During execution, QoS of actual services are measured and, in the case the actual values lead to violation of the QoS the concretisation phase is re-run;

SERENITY introduces the following events to monitor the activity of QoS-aware workflow engines: and generated by the workflow engine during the "concretisation" phase:

— *bind(procDefId, procId, activityName, service, port, op, serviceAddr)*: in the *procId* process instance the *op* operation of the *port* interface of the *service* abstract service will be executed by the service available at the address *serviceAddr.*

— *rebind(procDefId, procId, activityName, service, port, op, oldSAddr, newSAddr)*: in the *procId* process instance the *oldSAddr* service has been replaced by the *newSAddr* service for the execution of the the *op* operation of the *port* interface of the *service* abstract service.

— *unsatisfied(procDefId, procId, constrId, qosMeauseres)*: the *constrId* constraint defined on the *procDefId* process is no longer satisfied for the *procId* process instance given the *qosMeauseres* measures of QoS.

5.3. Event capture based on communication protocols

Many middleware systems are based on standardised communication protocols. As such, it is possible to capture events from the systems which make use of these middleware by observing the packets transmitted among processes/machines and analysing them accordingly to their protocol. For example, in a Web-Services [7] context, communication follows the SOAP [4] protocol standard. Thus, we can construct event collectors that capture the SOAP messages exchanged from different processes and translate these to invocations of web-services and the respective responses.

In this case there are two different positions where we can place our collectors; at the side of the web-services which are being called or at the side of the BPEL [1] engine which initiates these calls (if the calls have indeed been initiated by a BPEL engine). In both cases we wish to observe the events on the side of the web-services or on the side of the BPEL engine, so we need to install our proxies in a manner which will not cause any changes to the way the web-service/BPEL engine has been advertised to the rest of the world. Usually, web-services/BPEL engine are listening to the port #8080 of a server for requests from other applications. What we desire therefore is to introduce our proxy/event-collector in the server in such a way that all external requests to the web-service/BPEL engine will be received first by the proxy and only then will the proxy forward the requests to the web-service. We need therefore to change the port that the web-service/BPEL engine is listening to (say to port #8081) and attach our proxy to the port #8080. Once this is accomplished, our proxy

will be able to receive all requests from external processes (BPEL engines, other web-services) which wish to communicate with our web-service/BPEL engine, without the external processes knowing anything about the fact that we are capturing their requests. Once our proxy has noted the requests and notified the monitoring subsystem of them, it will forward them to the real port that the local web-service/BPEL engine is listening to, i.e., port #8081 in our example. External entities which wish to communicate with the web-service/BPEL engine continue to use the port 8080 since that was the one which was advertised for the web-service/BPEL engine. However, all their messages are captured by our proxy which is the one listening in port 8080 and once the proxy has informed the monitoring subsystem about them, it forwards them to the web-service/BPEL engine on port 8081, receives the reply from the web-service/BPEL engine, inform again the monitoring subsystem of it and forwards it to the external entity which had initiated the request.

5.3.1. Event capture on Tuple Space

Tuplespace based applications communicate by reading/writing tuples from/to a tuplespace. In order to monitor this kind of communication SERENITY requires that a tuplespace server a *Configuration* interface and use the *Logger* interface both defined in Section 5.1.

5.3.2. Capture of SOAP messages

SOAP messages can be transferred by a variety of protocols, such as HTTP or SMTP, but HTTP protocol has gained wider acceptance as it works well with today's Internet infrastructure. In our implementation we focused on capturing SOAP messages using the HTTP protocol for transportation. The HTTP protocol is a request/response protocol between a client and a server. A client initiates a request by establishing a connection to a particular port on a remote host. Upon receiving this request, the remote host sends back some status headers, such as “HTTP/1.1 200 OK” and a message, the body of which is the requested document. SOAP messaging works in a very similar way. Every SOAP request message is followed by a SOAP response in a synchronous exchange. The client creates a SOAP request message which binds in the HTTP request message (Table 5). Upon receiving this HTTP request the server which hosts the web service, extracts the SOAP request from the body of the message. The web service process the SOAP request and produces a SOAP response, which is send back attached in the body of the HTTP response message (Table 6).

```
POST /axis/servlet/AxisServlet HTTP/1.0
Content-Type: text/xml; charset=utf-8
Accept: application/soap+xml, application/dime, multipart/related, text/*
User-Agent: Axis/1.4
Host: localhost:8080
Cache-Control: no-cache
Pragma: no-cache
SOAPAction: ""
Content-Length: 438

<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
```

```
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"><soapenv:Body>

<ns1:getQuote soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:ns1="urn:xmltoday-delayed-quotes"><symbol
xsi:type="xsd:string">XXX</symbol></ns1:getQuote>

</soapenv:Body></soapenv:Envelope>
```

Table 5 – SOAP request through HTTP protocol

```
HTTP/1.1 200 OK

Content-Type: text/xml;charset=utf-8

Date: Mon, 11 Sep 2006 12:41:52 GMT

Server: Apache-Coyote/1.1

Connection: close


<?xml version="1.0" encoding="UTF-8"?>

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"><soapenv:Body><ns1:getQuoteResponse
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:ns1="urn:xmltoday-delayed-quotes"><getQuoteReturn
xsi:type="xsd:float">55.25</getQuoteReturn></ns1:getQuoteResponse></soapenv:Body></soape
nv:Envelope>
```

Table 6 - SOAP response through HTTP protocol

The above description of a synchronous SOAP messaging through HTTP protocol is the target of our Event Collector mechanism. The Event Collector as described in the section 0 works as a proxy between the client who requests a service and the server who offers the requested service. When a request for a service is made, the collector can capture the event and transmit it through a socket to the Event Receiver and then forward it to the real web service. In the same way the response of the web service is transmitted through the Event Collector to the client. In both cases the Event Collector reports the events of request and response to the Event Receiver in the form of XML documents based on the XML schema we defined in the Section 3.

— Implementation

The Event Collector is based on the Axis TCP Monitor (tcpmon) utility. This utility can be used to monitor the data flowing on a TCP connection. tcpmon can be placed in-between a client and a server. When a client makes a connection to tcpmon, tcpmon forwards the data to server along-with displaying it. The functionality of this utility has been extended in order to be able to (i) distinguish the request and response SOAP messages, (ii) construct the XML document for each event that a SOAP message represents and (iii) communicates these events to a remote Event Receiver. The

Event Receiver for our examples is a simple server application that can receive and display any incoming message (XML representations of events in our case).

— Required Software

The Event Collector has been developed and tested on a Debian GNU/Linux machine (kernel ver. 2.4.27-2-386). In order to demonstrate the tool the following software must be installed:

- Java 2 Platform, Standard Edition, v 1.4.2_12 (http://java.sun.com/j2se/1.4.2/download.html)
- Jakarta-Tomcat 5.0.30 (http://tomcat.apache.org/)
- Axis 1.4 (http://ws.apache.org/axis/)
- Xerces-J 2.8.0 (http://xerces.apache.org/xerces-j/)
- Bexee BPEL Execution Engine (http://bexee.sourceforge.net/)
- Apache Ant 1.6.5 (http://ant.apache.org/index.html)

Installation guides can be found in each site.

— Installation

The installation of the tool has two parts. In the first part we must setup the Event Receiver where the events are going to be reported.

- Untar the `EventReceiver.tgz` in your `$HOME` directory using the command "`tar -xzvf EventReceiver.tgz`".

- Go to the new directory called `EventReceiver` and execute the event receiver using the command "`java MultiEchoServer portnumber`". The portnumber should be the number of the port that the receiver will listen for new events. In our example if we want to use the port number 8008 then the command shall be "`java MultiEchoServer 8008`".

In the second part of the installation we must setup the Collector. The collector as described in the section 5.3 must be installed to the same server that provides the web service we would like to monitor. Also it should listen to the port number that our web service is already advertised. Thus, we must restart the Tomcat server giving him a new connection port. This change must be done at the file `conf/server.xml` in our Jakarta-Tomcat directory. There we must modify the Connection port as shown in the Figure 11. Let's say that the new port of Tomcat is set to 8081.

```
<!-- Define a non-SSL Coyote HTTP/1.1 Connector on port 8080 -->
<Connector port="8081"
           maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
           enableLookups="false" redirectPort="8443" acceptCount="100"
           debug="0" connectionTimeout="20000"
           disableUploadTimeout="true" />
<!-- Note : To disable connection timeouts, set connectionTimeout value
 to 0 -->
```

Figure 11 – Modifing connection port in Tomcat

After that we have to

- untar the archive called `tcpmon.tgz` using the command: `tar –xzvf tcmpon.tgz`.

- A new directory has been created with the name `tcpmon-1.0-src`. In order to execute the event collector the `CLASSPATH` must be updated. Include the path `/tcpmon-1.0-src/src/` to your existing `CLASSPATH`.

- The application can now be executed by the command: "`java org.apache.ws.commons.tcpmon.TcpTunnel 8080 localhost 8081 EventReceiverHost 8008`". The first argument is the port number that the collector should listen. The second and third arguments are the hostname and the port number of the server where the collector should forward the requests. As we mentioned above the collector is running in the same server as the web service (localhost) and our web service is listening at 8081. Finally the last two arguments are the remote server where the Event Receiver waits for the events.

After that every SOAP message that reaches our server in the port 8080, is displayed by the tcpmon tool (Figure 13), reported to the event receiver (Figure 12) and forwarded to the real web service that operates in the port 8081.

```
Received: <?xml version="1.0" encoding="UTF-8"?>
Received: <event>
Received:     <eventID>12</eventID>
Received:     <type>
Received:         <OperationMessage>
Received:             <operationName>getInfo</operationName>
Received:             <operationID>7</operationID>
Received:             <status>request</status>
Received:             <op_args>
Received:                 <argument>
Received:                     <SimpleArgument>
Received:                         <name>symbol</name>
Received:                         <type>string</type>
Received:                         <value>IBM</value>
Received:                         <argumentType>IN</argumentType>
Received:                     </SimpleArgument>
Received:                 </argument>
Received:                 <argument>
Received:                     <SimpleArgument>
Received:                         <name>info</name>
Received:                         <type>string</type>
Received:                         <value>address</value>
Received:                         <argumentType>IN</argumentType>
Received:                     </SimpleArgument>
Received:                 </argument>
Received:             </op_args>
Received:         </OperationMessage>
Received:     </type>
Received:     <message_args>
Received:         <sender>
Received:             <ipAddress>127.0.0.1</ipAddress>
Received:         </sender>
Received:         <receiver>
Received:             <name>/axis/servlet/AxisServlet</name>
Received:             <ipAddress>127.0.0.1</ipAddress>
Received:             <port>8080</port>
Received:         </receiver>
Received:         <context>
Received:             <collectionTime>2006-09-11T18:42:04</collectionTime>
Received:             <reportTime>2006-09-11T18:42:04</reportTime>
Received:             <eventSource>
Received:                 <name>/axis/servlet/AxisServlet</name>
Received:                 <ipAddress>127.0.0.1</ipAddress>
Received:                 <port>8080</port>
Received:             </eventSource>
Received:         </context>
Received:     </message_args>
Received: </event>
```

Figure 12 - Event Receiver

```
POST /axis/servlet/AxisServlet HTTP/1.0
Content-Type: text/xml; charset=utf-8
Accept: application/soap+xml, application/dime, multipart/related, text/*
User-Agent: Axis/1.4
Host: localhost:8080
Cache-Control: no-cache
Pragma: no-cache
SOAPAction: ""
Content-Length: 462
Authorization: Basic dXNlcjM6cGFzczM=

<?xml version="1.0" encoding="UTF-8"?><soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/enve
lope/" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"><
soapenv:Body><ns1:getInfo soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xmlns:ns1="urn:
cominfo"><symbol xsi:type="xsd:string">IBM</symbol><info xsi:type="xsd:string">address</info></ns1:getInfo>
</soapenv:Body></soapenv:Envelope>




HTTP/1.1 200 OK
Content-Type: text/xml;charset=utf-8
Date: Mon, 11 Sep 2006 18:45:56 GMT
Server: Apache-Coyote/1.1
Connection: close

<?xml version="1.0" encoding="utf-8"?><soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/enve
lope/" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"><
soapenv:Body><ns1:getInfoResponse soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xmlns:n
s1="urn:cominfo"><getInfoReturn xsi:type="soapenc:string" xmlns:soapenc="http://schemas.xmlsoap.org/soap/en
coding/">Armonk, NY</getInfoReturn></ns1:getInfoResponse></soapenv:Body></soapenv:Envelope>
```

Figure 13 - Event Collector

— Example

In order to demonstrate the features of the Event Collector, in this section we use two examples. The first example is a use case taken from the samples of Axis1.4 and it can be found in the directory `$HOME/axis-1.4/samples/stock`. Before we execute this example we must make sure that some preconditions are valid:

- The directories `$HOME/axis-1.4/` and `$HOME/axis-1.4/samples/` are already included in the `CLASSPATH`. If not they can be added with the following commands:

```
export CLASSPATH=$CLASSPATH:$HOME/axis-1.4/
export CLASSPATH=$CLASSPATH:$HOME/axis-1.4/samples/
```

- The sample stock web service must be deployed and undeployed on the server on the specific port number. In our implementation we changed the connection port number of our server to 8081. The sample code contained in the file `$HOME/axis-1.4/samples/stock/testit.sh` will try to deploy and undeploy the web service in the default port number, which is 8080. Therefore for the right execution of the sample we can either edit the `testit.sh` file adding the argument for the new port number on the deploy

and undeploy commands (Figure 14) or execute the sample giving the following series of commands:

- Deploy web service: `java org.apache.axis.client.AdminClient -p 8081 deploy.wsdd`
- Execute sample invocations to the service: `java samples.stock.GetQuote -uuser1 -wpass1 XXX`
- `java samples.stock.GetQuote -uuser2 XXX`
- `java samples.stock.GetInfo -uuser3 -wpass3 IBM address`
- Undeploy service: `java org.apache.axis.client.AdminClient -p 8081 undeploy.wsdd`

```
  GNU nano 1.2.4                File: testit.sh                     Modified

#!/bin/sh
# this assumes webserver is running on port 8081

echo "Deploy everything first"
java org.apache.axis.client.AdminClient -p 8081 deploy.wsdd $*

echo "These next 3 should work..."
java samples.stock.GetQuote -uuser1 -wpass1 XXX $*
java samples.stock.GetQuote -uuser2 XXX $*
java samples.stock.GetInfo -uuser3 -wpass3 IBM address

echo "The rest of these should fail... (nicely of course)"
java samples.stock.GetQuote XXX $*
java samples.stock.GetQuote -uuser1 -wpass2 XXX $*
java samples.stock.GetQuote -uuser3 -wpass3 XXX $*

echo "This should work but print debug info on the client and server"
java samples.stock.GetQuote -d -uuser1 -wpass1 XXX $*

# Now undeploy everything
java org.apache.axis.client.AdminClient -p 8081 undeploy.wsdd $*



^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Page ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is  ^V Next Page ^U UnCut Txt ^T To Spell
```

Figure 14 - Modification of example script

During the execution of the invocations to the web service we will be able to see the events generated and transmitted to the Event Receiver while the SAOP messages exchanged are displayed to the Event Collector.

With the second example we demonstrate the event collection from the execution of a BPEL process. The sample BPEL process is from the Bexee BPEL execution engine and can be found in the `$HOME/bexee-0.1/samples/BookTravel/TravelProcess/` directory. For the same reasons as above the deployment and undeployment of the web services that this process uses must be done to the 8081 port. Thus, we must edit the `built.xml` file located in `$HOME/ bexee-0.1/samples/BookTravel/TravelProcess/` and adjust the port number in the sections referring to the deployment and undeployment of the process as shown in Figure 15.

```
  GNU nano 1.2.4                    File: build.xml                    Modified

<project name="bexee-ant" default="deploy" basedir=".">

        <!-- Project structure - should not have to be modified -->
        <property name="src.dir" value="." />
        <property name="lib.dir" value="../../../lib" />

        <!-- CLASSPATH -->
        <path id="classpath">
                <fileset dir="${lib.dir}">
                        <include name="**/*.jar" />
                </fileset>
        </path>

        <!-- Declare bexee Ant Tasks -->
        <taskdef resource="bexee-tasks.properties" classpathref="classpath" />

        <!-- Deploy Target -->
        <target name="deploy">
                <echo>${lib.dir}</echo>

                <bexee-deploy
                        bpel="${src.dir}/BookTravel.bpel"
                        wsdl="${src.dir}/BookTravel.wsdl"
                        url="http://localhost:8081/bexee/services/Manager">
                        <partnerwsdl dir="${src.dir}" >
                                <include name="TravelService.wsdl" />
                        </partnerwsdl>
                </bexee-deploy>

        </target>

        <!-- Undeploy Target -->
        <target name="undeploy">

                <bexee-undeploy
                        name="TravelProcess"
                        url="http://localhost:8081/bexee/services/Manager" />

        </target>

        <!-- Start Target -->
        <target name="start">

                <bexee-start
                        url="http://localhost:8080/bexee/services/TravelProcess"
                        operation="initiate"
                        input="rucki-zucki" />

        </target>

</project>
```

Figure 15 - Modification of the built.xml

After that we can use the commands:

- `ant deploy`: for the deployment of the service
- `ant start`: to invoke the process
- `ant undeploy`: to undeploy the service.

5.3.2.1 Implementation Limitations/Benefits of Capturing SOAP messages

Currently our implementation is limited in capturing SOAP messages based on the synchronous SOAP over HTTP message exchange. HTTP messaging protocol consists of a response message for each request message during a HTTP connection. Therefore every SOAP request shall have a response message. Our implementation is capable to capture and correlate the request/response messages even if other invocations have taken place in the time between.

In some cases the web service is possible to respond with a Fault message. This can happen when the request message is faulty (e.g. bad method parameter values, improperly formatted) or for other back-end problems. In these cases the response message has a different structure that informs the client for the faults that had been encountered. For now our collector can not distinguish this kind of messages.

Finally SOAP messages are also exchanged by BPEL engines. BPEL engines can orchestrate existing Web Services, by defining interactions among them. By this way BPEL engines can offer new services by composing and integrating existing services. Any BPEL process-a composition of web services- is appeared to the end user as a single web service. When the user requests the execution of such a service the BPEL engine executes a number of requests to other web services in order to compose the response to the request. In our implementation of Event Collector we took into account such processes and it is possible to distinguish and collect events produced by such BPEL engines.

6. Event Capturing at the Application Layer

6.1. Aspect based capture

This section gives a preliminary report into the event capturing mechanism we plan to develop for monitoring binary executables for which we do not have the source code and which have been developed with a language other than Java. The problem in this case is that there is usually extremely little information one can derive from the binary code and it is difficult to control it at specific locations. That is, one cannot easily create a mechanism for stopping an application when a particular internal function is called or some variable changes its value. Indeed, most of the time the information which assigns names to the internal variables and functions has been lost at this stage. This is the basic reason why there has been a lot of work on Aspect-Oriented Programming (AOP) for Java, where the bytecode form of a program contains all the information we need, while there's extremely little work on AOP for C/C++ binaries where this information is traditionally removed (mostly for optimisation reasons).

The basic design goal of AOP is to weave (i.e., introduce) additional code into an application at particular points (usually before and after some function call). So it needs to be able to identify a particular function/method F, and introduce a new function call A before F is executed and another function B after F has been executed. In order to be able to identify a specific function (or variable) in a binary executable, we need to have what is commonly called debugging information for that binary. That is, the binary should have been compiled with the appropriate flags for producing extra information which is used during debugging. This information is always present in a normal Java bytecode file and contains the names of the functions, variables, etc. that form the program. It also contains information which allows debugging tools to stop the execution of the program temporarily at various locations so that the developers can examine the internal state of the program. In the Linux OS that we are mainly working on, the default compiler and debugger is GCC [11] & GDB [12] respectively. When passed the –g option GCC produces the extra debugging information needed by GDB. The latter can then be used to set so called breakpoints, i.e., places in the code where the execution of the code should be stopped. These breakpoints can be at the entrance to a function or even at a specific source line. Once the program is stopped, the developer can use the front-end of GDB to examine the state of the system, call other functions, etc. Another useful option of GDB is its ability to set watch-points, i.e., conditions upon which the program should be stopped. These conditions are defined with respect to the values of the program's variables.

Therefore, one can use this mechanism to build an AOP-like tool for C/C++ binary programs, where the aspects are introduced through breakpoints and watchpoints. A very simple such prototype can communicate with GDB using its textual interface, passing it commands to execute and receiving the results as strings that it then parses. Indeed this is how some of the interfaces to GDB currently work (e.g., DDD [8], Emacs, GDBtk).

Some of the disadvantages of this approach are:

— It is very sensitive to changes in GDB's output.

— Performance is restricted by the speed of communication between the GUI and GDB.

— It is difficult to keep the GUI [4] consistent with the CLI [5].

However, there is an ongoing project for exposing the internal functionality of GDB as a normal library (libgdb, see [13]). Using this library one can develop the AOP-like tool we envisage programmatically, without having to send commands and receive results as strings.

Using this tool we will be able to stop a program at specific locations (in our case at function calls), call some other code (our event collectors) and then restart the application. We will be able to observe internal function calls and variables, calls to the middleware, system calls, signals, exceptions, etc. The only problem is that this tool will only work for programs compiled with the GCC compiler and then only if they have been compiled with the debugging information enabled (i.e., the –g switch).

[4] Graphical User Interface

[5] Command-Line Interface

7. Conclusion & Future Work

In this document we have described the basic set of information collection mechanisms that we have developed in the SERENITY project in order to support run-time security and dependability monitoring**.** These mechanisms support the collection of events from:

— Operating System Calls, and

— Middleware including mechanisms for collection of events from tuple spaces and communication protocols

To provide a uniform way of reporting the events captured by the above mechanisms we have defined an XML schema which we have discussed in this report.

The implementations of the above event collection mechanisms are provided as a source code archive that accompanies this deliverable. This archive includes also simple applications that can be used to deploy the developed mechanisms in order to demonstrate their use. The basic instructions for the installation and use of these applications have been given in this deliverable.

Additional mechanisms to support the collection of events from application layers will be specified and implemented in the next deliverable on event collection mechanisms (A4.D2.4) that is due in month 15. As part of the investigation of possible mechanisms for event collection at the application layer we will experiment with:

— Aspect-based event collection

— Java bytecode instrumentation

— Source code instrumentation

Appendix A. XML Schema of Events

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

    <xs:complexType name="eventType">
        <xs:sequence>
            <xs:element name="eventID" minOccurs="1"
                        maxOccurs="1" type="EventID"/>
            <xs:element name="type" minOccurs="1"
                        maxOccurs="1" type="EventType"/>
            <xs:element name="message_args"  type="MessageArgsList"
                        minOccurs="1" maxOccurs="1"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="MessageArgsList">
        <xs:sequence>
            <xs:element name="sender" type="Entity"
                        minOccurs="1" maxOccurs="1" />
            <xs:element name="receiver" type="Entity"
                        minOccurs="1" maxOccurs="1" />
            <xs:element name="context" type="Context"
                        minOccurs="1" maxOccurs="1" />
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name='Entity'>
        <xs:sequence minOccurs="1" maxOccurs="1">
            <xs:element name="name" type="xs:string"
                        minOccurs="0" maxOccurs="1"/>
            <xs:element name="ipAddress" type="ip"
                        minOccurs="0" maxOccurs="1"/>
            <xs:element name="port" type="xs:long"
                        minOccurs="0" maxOccurs="1"/>
            <xs:element name="UserID" type="xs:long"
```

```
                    minOccurs="0" maxOccurs="1"/>
            <xs:element name="processID" type="xs:long"
                    minOccurs="0" maxOccurs="1"/>
        </xs:sequence>
    </xs:complexType>

    <xs:simpleType name="ip">
        <xs:restriction base="xs:string">
            <xs:pattern value = "[0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}"/>
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="Status">
        <xs:restriction base ="xs:string">
            <xs:pattern value = " REQ-B|REQ-A|RES-B|RES-A"/>
        </xs:restriction>
    </xs:simpleType>

    <xs:complexType name="EventType">
        <xs:choice minOccurs="1" maxOccurs="1">
            <xs:element name="OperationMessage" type="OpMsg"/>
            <xs:element name="DataMessage" type="DataMsg"/>
        </xs:choice>
    </xs:complexType>

    <xs:complexType name='OpMsg'>
        <xs:sequence minOccurs="1" maxOccurs="1">
            <xs:element name="operationName" type="xs:string"
                    minOccurs="1" maxOccurs="1"/>
            <xs:element name="operationID" type="xs:long"
                    minOccurs="1" maxOccurs="1"/>
            <xs:element name="status" type="Status"
                    minOccurs="1" maxOccurs="1" />
            <xs:element name="op_args" type="argumentsType"
                     minOccurs="0" maxOccurs="1"/>
        </xs:sequence>
    </xs:complexType>
```

```
    <xs:complexType name="DataMsg">
        <xs:sequence>
            <xs:element name="type" type="xs:string"
                        minOccurs="1" maxOccurs="1"/>
            <xs:element name="value" type="xs:string"
                        minOccurs="1" maxOccurs="1"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="Context">
        <xs:sequence>
            <xs:element name="collectionTime" type="xs:dateTime"
                        minOccurs="1" maxOccurs="1"/>
            <xs:element name="reportTime" minOccurs="1"
                        maxOccurs="1" type="xs:dateTime"/>
            <xs:element name="eventSource" type="Entity"
                        minOccurs="1" maxOccurs="1" />
        </xs:sequence>
    </xs:complexType>

    <xs:simpleType name="EventID">
        <xs:restriction base="xs:unsignedLong"/>
    </xs:simpleType>

    <xs:complexType name="argumentsType">
        <xs:sequence minOccurs="1" maxOccurs="unbounded">
            <xs:element type="argumentType" name="argument"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="argumentType">
        <xs:choice maxOccurs="1">
            <xs:element name="SimpleArgument" type="simpleArgument"
                        minOccurs="1" maxOccurs="1"/>
            <xs:element name="Struct" minOccurs="1" maxOccurs="1">
                <xs:complexType>
                    <xs:sequence minOccurs="1" maxOccurs="unbounded">
                        <xs:element type="argumentType" name="variable"/>
```

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

    <xs:complexType name="simpleArgument">
        <xs:sequence minOccurs="1" maxOccurs="1">
            <xs:element name="name" type="xs:string"
                        minOccurs="1" maxOccurs="1"/>
            <xs:element name="type" type="xs:string"
                        minOccurs="1" maxOccurs="1"/>
            <xs:element name="value" type="xs:anySimpleType"
                        minOccurs="1" maxOccurs="1"/>
            <xs:element name="argumentType" type="xs:string"
                        minOccurs="1" maxOccurs="1"/>
        </xs:sequence>
    </xs:complexType>

</xs:schema>
```

Table 7 – Event representation XML schema

Appendix B. JavaSpace operations mapping

This appendix introduces the mapping between the tuplespace operations defined by the JavaSpace Specifications and the general tuplespace model described in paragraph 5.1.2.

JavaSpace	Abstract Tuplespace Model

`Lease write(Entry entry,`
`Transaction txn,`
`long lease)`
	out(entry)

`Entry read(Entry tmpl,`
`Transaction txn,`
`long timeout)`
	rd(tmpl)

`Entry readIfExists(Entry tmpl,`
`Transaction txn,`
`long timeout)`
	rdp(tmpl)

`Entry take(Entry tmpl,`
`Transaction txn,`
`long timeout)`
	in(tmpl)

`Entry takeIfExists(Entry tmpl,`
`Transaction txn,`
`long timeout)`
	inp(tmpl)

`Collection take(Collection tmpls,`
`Transaction txn,`
`long timeout,`
`long maxEntries)`
	rew(tmpl, ***null****)*

`MatchSet contents(Collection tmpls,`
`Transaction txn,`
`long leaseDuration,`
`long maxEntries)`
	copy_collect(tmp)

`List write(List entries,`
`Transaction txn,`
`List leaseDurations)`
	*rew(****null****, entries)*

Table 8 - JavaSpace operations Mapping

References

[1] Andrews T., Curbera F., Dholakia H., Goland Y., Klein J., Leymann F., Liu K., Roller D., Smith D., Thatte S., Trickovic I., and Weerawarana S. (2005) "Business process execution language for web services version 1.1" Technical report, BEA Systems, International Business Machines Corporation, Microsoft Corporation, SAP AG, Siebel Systems, July 2002. Last updated: February. Available from http://www.ibm.com/developerworks/library/ws-bpel/

[2] Andrieux A., Czajkowski K., Dan A., Keahey K., Ludwig H., Pruyne J., Rofrano J., Tuecke S. and Xu M. (2004) "Web Services Agreement Specification", Global Grid Forum, May. Available from: http://www.gridforum.org/Meetings/GGF11/Documents/draft-ggf-graap-agreement.pdf

[3] Bartel M., Boyer J., Fox B., LaMacchia B., and Simon E. (2002). XML-signature syntax and processing, In D. Eastlake, J. Reagle, and D. Solo, editors, W3C Recommendation. Feb. 12. {Last accessed: August 30, 2006}, http://www.w3.org/TR/xmldsig-core/.

[4] Box D., Ehnebuske D., Kakivaya G., Layman A., Mendelsohn N., Nielsen H. F., Thatte S., and Winer D. (2000) "Simple Object Access Protocol (SOAP)". Technical Report version 1.1, W3C, May. Available from http://www.w3.org/TR/soap/

[5] Busi N., Ciancarini P., Gorrieri R., Zavattaro G. (2001) "Coordination Models: A Guided Tour", in *Coordination of Internet Agents*, A. Omicini, M. Klusch, R. Tolksdorf, F. Zambonelli (eds.), Springer.

[6] Campadello S. et al. (2006) "S&D Requirements Specification", SERENITY Deliverable A7.D2.1

[7] Christensen E., Curbera F., Meredith G., and Weerawarana S. (2001) "Web services description language (WSDL) 1.1". Technical Report Note 15, W3C, March. Available from http://www.w3.org/TR/wsdl

[8] DDD (2005) "DDD - Data Display Debugger" Retrieved from http://www.gnu.org/software/ddd/ Last updated: 2005-10-22

[9] Freeman E., Hupfer S., Arnold K. (1999) *JavaSpaces: Principles, Patterns, and Practice*, The Jini Technology Series, Addison-Wesley.

[10] Gelernter, D. (1985). "Generative communication in Linda". ACM Transactions on Programming, 2(1):80—112.

[11] GNU-GCC (2006) "GCC, the GNU Compiler Collection" Retrived from http://gcc.gnu.org/ Last updated: 2006-07-25

[12] GNU-GDB (2006) "GDB: The GNU Project Debugger" Retrieved from http://sources.redhat.com/gdb/ Last updated: 2006-07-05

[13] GNU-libgdb (2002) "libGDB" Retrieved from http://sources.redhat.com/gdb/papers/libgdb2/ Last updated: 2002-09-19

[14] Imamura T., Dillaway B. and Simon E. (2002) XML Encryption Syntax and Processing, In D. Eastlake and J. Reagle., editors, W3C Proposed Recommendation, Dec. 10. {Last accessed: August 30, 2006}, http://www.w3.org/TR/xmlenc-core/

[15] Nanda M.G., Chandra S., Sarkar V. (2004) "Decentralizing Execution of Composite Web Services", OOPSLA'04, Oct. 24-28.

[16] Pernici B. (2006) (ed.), *Mobile Information Systems*, Sprinter.

[17] Rossi D., Cabri G., Denti E. (2001) "Tuple-based Technologies for Coordination", in *Coordination of Internet Agents*, A. Omicini, M. Klusch, R. Tolksdorf, F. Zambonelli (eds.), Springer.

[18] Rowstron A. (2001) "Run-Time Systems for Coordination", in *Coordination of Internet Agents*, A. Omicini, M. Klusch, R. Tolksdorf, F. Zambonelli (eds.), Springer.

[19] Shanahan M. P. (1999) The Event Calculus Explained, in Artificial Intelligence Today, ed. M.J.Wooldridge and M.Veloso, Springer Lecture Notes in Artificial Intelligence no. 1600,), pages 409-430, Springer.

[20] Wickoff P. (1998) "T Spaces", IBM System Journal, Vol.37, No.3.

[21] Wikipedia contributors (2006) Battleship (game). In *Wikipedia, The Free Encyclopedia,* last revised 13 September 2006, 10:27 UTC, Accessed 13 September 2006 <http://en.wikipedia.org/w/index.php?title=Battleship_%28game%29&oldid=75479522>

[22] Zeng L., Benatallah B., Ngu H.H., Dumas M., Cakagnanam J., Chang H. (2004) "QoS-aware Middleware for Web Services Composition", IEEE Transactions On Software Engineering, Vol.30, No.5, May.