A2.D2.3 – Enhanced specification language for workflow S&D requirements/properties

S. Kokolakis, C. Rudolph, Z. Velikova

Document Number	A2.D2.3
Document Title	Enhanced specification language for workflow S&D requirements/properties
Version	1.0
Status	Final
Work Package	WP2.2
Deliverable Type	Report
Contractual Date of Delivery	30 June 2008
Actual Date of Delivery	
Responsible Unit	UNA
Contributors	SIT, SAP
Keyword List	S&D Requirements, Workflow, Web Services, APA, WS-BPEL
Dissemination level	PU

Change History

Version	**Date**	**Status**	**Author (Unit)**	**Description**
0.1	Sep 01, 2008	Draft	C. Rudolph, Z. Velikova (SIT)	First complete draft
1.0	Sep 07, 2008	Final	S. Kokolakis (UNA)	Final draft

Executive summary

This report supplements deliverable A2.D2.1 ”S&D requirements for workflows” and extends deliverable A2.D2.2 “Revised specification language for workflow S&D requirements/properties”. In A2.D2.1 a requirements specification language has been defined, titled ’SecureBPEL’, which aims to support security engineers in their effort to identify S&D requirements for workflows. Therefore, this language is abstract, easy to read and comprehend, and can be integrated with the process definition language BPEL. Nevertheless, this language is not appropriate for formal analysis of processes.

In A2.D2.2 a language for the formal definition of S&D properties was provided, which uses a process-oriented model with an operational semantics. Workflows (e.g. described with BPEL) can be translated into this model. Thus, security and dependability requirements can be formally defined for complete workflows, parts of a workflow, particular security-relevant phases of a workflow or for single services. The underlying formalism is the same that was used in activity A3 to define security and dependability for networks and devices. Thus, we are able to use the security analysis and verification platform (SHVT), which has already successfully been applied in SERENITY Activity A3, for the security analysis and verification of workflows.

This report is an extended version of A2.D2.2 with additional workflow requirements. In addition to the generic requirements formallized in A2.D2.2, more concrete workflow properties are formalized. Moreover, this version is inline with the Formal S&D Properties Language (FPL), the generic properties language to be used for the selection of S&D Classes and Patterns.

Table of Contents

1. Introduction **1**

2. Translating WS-BPEL specifications to formal, executable automata models **2**
2.1. Introduction . . . 2
2.2. Background Knowledge . . . 2
2.2.1. WS-BPEL . . . 2
2.2.2. APA . . . 6
2.3. Translating a WS-BPEL Process into APA . . . 7
2.3.1. Data Type Definitions . . . 7
2.3.2. State Transitions and Roles in the APA Model . . . 10
2.3.3. A WS-BPEL Workflow Example . . . 11

3. Security requirements for workflows **14**
3.1. Introduction . . . 14
3.2. System behaviour specification and agents' knowledge about a system . . . 15
3.3. Agents' view and knowledge about the global system behaviour . . . 15

4. Examples for security requirements specification for workflows and services **18**
4.1. Authenticity . . . 18
4.1.1. Definition of the property . . . 18
4.1.2. Examples . . . 18
4.2. Proof of authenticity . . . 19
4.2.1. Definition of the property . . . 19
4.2.2. Examples . . . 19
4.3. A remark to integrity . . . 19
4.4. Confidentiality . . . 20
4.4.1. Definition of the property . . . 20
4.5. Enforcing certain system behaviour . . . 21
4.6. Refined workflow requirements . . . 21
4.6.1. Requirements for single services . . . 23
4.6.2. Requirements for workflow processes . . . 25

5. Conclusions **26**

Appendix A. BPEL **27**

Appendix B. WSDL **30**

Appendix C. APA Definition **32**

1. Introduction

A wide variety of security requirements can occur for workflows and even more for distributed workflows in AmI environments. Such requirements can be, for example, concerned with the authenticity of an entity performing a particular service in a workflow, integrity or confidentiality of data that is transported between entities involved in the workflow, or the enforcement of particular (distributed) sequences of actions (or services) in the workflow. A combination of security mechanisms for single services as well as for the overall workflow can be required to satisfy all these different security properties. The formal framework for secuirty properties introduced by Gürgens, Ochsenschläger and Rudolph [11] provides the expressiveness and flexibility to cover most of these properties. Furhtermore, this framework is also underlying the property specification language for networks and devices in A3. Thus, by applying htis framework to workflows and services, the required variety of properties can be adequately described and properties on the different levels of workflows and devices and network communication can be easily related and analysed using the same toolset.

This report is structured as follows. First, a formal operational semantics for BPEL specifications is exemplarily defined by translating WS-BPEL descriptions to executable APA (asynchronous product automata) models. This is a necessary, and critical, step towards the formal specification and verification of properties for workflows. These models can be executed and validated within the SH Verification Tool. Furthermore, security analysis approaches for the SH Verification Tool [12, 13] can also be applied. Then, the formal framework for secuirty properties is applied to workflow specifications and finally a few examples of typical workflow and services security requirements are given.

2. Translating WS-BPEL specifications to formal, executable automata models

2.1. Introduction

The Web Services Business Process Execution Language (WS-BPEL) [2] is a language for describing the behavior of business processes based on web services. A WS-BPEL process specifies the interactional behavior of this process with other processes, the so called partners. The interaction between processes may be nontrivial. Despite its wide acceptance, WS-BPEL provides no support for the detection of possible deadlocks or other interaction errors or even security properties. There exist different reasons for improper process interaction such as:

— A process may have an erroneous design.

— The interactional behaviors of two processes exclude each other.

Thus, it is crucial to be able to test whether all processes interact properly. A formal model that can be used for this type of testing can also be a basis for exact specification of security properties. In this work we present a model for a WS-BPEL process, which translates a workflow WS-BPEL definition into an asynchronous product automata (APA). We identify all possible actions and objects in the workflow by specifying a set of possible state transition sequences and state components, which uniquely identify the behavior of the corresponding APA. We propose a formal model for WS-BPEL workflows that can specify in details the order in which the different webservices are invoked and the exchange of messages between the workflow partners.

The rest of this section is structured as follows. In the following section we give a general and formal definition of APA and briefly introduce the most common activities and objects, specified in the WS-BPEL Specification [2]. In section 2.3. we will give a detailed description of the translation of the WS-BPEL elements into data types and transition patterns, suitable for an APA definition in SHVT[6]. We will formally define these data types and we will translate them into the syntax used to define APAs in SHVT. We will also present a small WS-BPEL workflow "SearchDoctor" and its model. The corresponding to the "SearchDoctor" BPEL, WSDL and APA definitions can be found in the Appendix.

2.2. Background Knowledge

2.2.1. WS-BPEL

The Web Services Business Process Execution Language (WS-BPEL) [2] is a language for describing the behavior of business processes based on web services. For the specification of a business process, WS-BPEL provides activities and distinguishes between basic and structured activities.

Basic Activities

— **synchronous `<invoke>` (request-reply)**

```
<invoke name          = "sampleName"
        partnerLink   = "samplePartnerLink"
        portType      = "samplePortType"
```

```
        operation     = "sampleOperation"
        inputVariable = "sampleInput"
        outputVariable= "sampleOutput"/>
```

Invoking an operation on web services provided by partners is a basic activity. A synchronous invocation requires both an input variable and an output variable. The attributes for the `invoke` activity can be listed as:

- `name` a local identifier of the invoke operation
- `partnerLink` identifies the partner link used to connect to the service.
- `portType` identifies the port type(optional - can be implied by the `partnerLink` and `partnerRole` in that partner link).
- `operation` identifies the service operation to invoke.
- `inputVariable` identifies a variable that contains the business data being sent to the service.
- `outputVariable` identifies a variable that receives the business data being returned from the service

— **asynchronous <invoke> one-way**
An asynchronous invocation requires only the input variable of the operation because it does not expect a response as part of the operation.

```
<invoke name          = "sampleName"
        partnerLink   = "samplePartnerLink"
        portType      = "samplePortType"
        operation     = "sampleOperation"
        inputVariable = "sampleInput"/>
```

— **<receive>**
The `receive` activity waits for a message.

```
<receive name        = "sampleName"
         partnerLink = "samplePartnerLink"
         portType    = "samplePortType"
         operation   = "sampleOperation"
         variable    = "sampleVariable"/>
```

The `receive` activity specifies the `partnerLink` it expects to receive from, the `portType` and `operation` that it expects the partner to invoke. In addition, it may specify a `variable` that is to be used to receive the message data received.

— **<reply>**
A reply activity is used to send a response to a request previously accepted through a `receive` activity. Such responses are only meaningful for synchronous interactions.

```
<reply name        = "sampleName"
       partnerLink = "samplePartnerLink"
       portType    = "samplePortType"
       operation   = "sampleOperation"
       variable    = "sampleVariable"/>
```

A optional attribute `variable` that contains the message data to be sent in reply may be specified.

— **<assign>**
The `assign` activity is used to copy data from one variable to another (with type-compatible values) and to construct and insert new data using expressions.

```
<assign name = "sampleName">
        <copy>
              from-spec
              to-spec
        </copy>
</assign>
```

The source $from - spec$ must be one of the following forms:

- `<from variable="sampleVariable" part="samplePart"?/>`
- `<from partnerLink="sampleLink" endpointReference="myRole|partnerRole"/>`
- `<from variable="sampleVariable" property="sampleProperty"/>`
- `<from expression="general-expr"/>`
- `<from> ... literal value ... </from>`

In the first $from - spec$ variants the `variable` attribute provides the name of a variable and `part` is an optional attribute used to provide the name of a part within a variable of a WSDL message type. In the second variant the `partnerLink` attribute value is the name of a partnerLink declared in the process and if the value of `endpointReference` is "$myRole$" then the source is the endpoint reference of the process with respect to that `partnerLink`; if its value is "partnerRole" then the source is the partner's endpoint reference for the `partnerLink`. The target $to - spec$ must be one of the following forms:

- `<to variable="ncname" part="ncname"?/>`
- `<to partnerLink="ncname"/>`
- `<to variable="ncname" property="qname"/>`

The attributes and the use of these variants $to - spec$ is equivalent to the corresponding $from - spec$ variants.

Structured Activities

— **<while>**
The `while` activity supports repeated performance of a specified iterative activity.

```
<while condition="bool-expr">
   activities
</while>
```

— **<switch>**
The switch activity consists of an ordered list of one or more conditional branches defined by case elements, followed optionally by an otherwise branch. The first branch whose condition holds true is taken and provides the activity performed for the switch.

```
<switch>
       <case condition="bool-expr">
           activity
       </case>
       <otherwise>
           activity
       </otherwise>
</switch>
```

— **<pick>**
The pick activity awaits the occurrence of one of a set of events and then performs the activity associated with the event that occurred. The occurrence of the events is often mutually exclusive. In common case events, for which pick activity waits, are onMessage and onAlarm The onMessage tag indicates that the event specified is an event that waits for a message to arrive. The interpretation of this tag and its attributes is very similar to a receive activity. The onAlarm tag marks a timeout event. The for attribute specifies the duration after which the event will be signaled. The clock for the duration starts at the point in time when the associated scope starts.

```
<pick>
     <onMessage partnerLink="ncname"
                portType="qname"
                operation="ncname"
                variable="ncname">
           activity
     </onMessage>
     <onAlarm (for="duration-expr" | until="deadline-expr")>
           activity
     </onAlarm>
</pick>
```

Variables and Partner Links

— **<variable>**
A variable is a data container for WSDL message, an XML Schema or an XML Schema element.

```
<variable name        = "sampleVName"
          messageType = "sampleVMType"
          type        = "sampleVType"
          element     = "sampleVElement"/>
</variable>
```

The attributes for the variable activity can be listed as:

- `name` a local identifier which should be unique within its scope.
- `messageType` refers to a WSDL message type definition.
- `type` refers to an XML Schema simple type.
- `element` refers to an XML Schema element.

— **`<partnerLinkType>`**
A `partnerLinkType` characterizes the conversational relationship between two services by defining the "roles" played by each of the services in the conversation and specifying the `portType` provided by each service to receive messages within the context of the conversation. Each role specifies exactly one WSDL portType.

```
<partnerLinkType name = "sampleName">
    <role name = "sampleRoleName1">
        <portType name = "samplePTName1"/>
    <role/>
    <role name = "sampleRoleName2">
        <portType name = "samplePTName2"/>
    <role/>
<partnerLinkType/>
```

Alternatively, using similar syntax the partner link type definition can be placed within the WSDL document.

```
<plnk:partnerLinkType name="sampleLTName">
    <plnk:role name="sampleRoleName1">
        <plnk:portType name="samplePTName1"/>
    </plnk:role>
    <plnk:role name="sampleRoleName2">?
        <plnk:portType name="samplePTName2"/>
    </plnk:role>
</plnk:partnerLinkType>
```

2.2.2. APA

Asynchronous product automata (APA) are a universal and very flexible operational description concept for cooperating systems [21]. They "naturally" emerge from formal language theory [20].Their specification, analysis and verification is supported by the SH-verification tool (SHVT) [6]. SHVT provides components for the complete cycle from formal specification to exhaustive validation. [21].

An APA can be seen as a family of elementary automata. The set of all possible states of the whole APA is structured as a product set; each state is divided into state components. In the following we will refer to the set of all possible states as state set. The state sets of elementary automata consist of components of the state set of the APA. Different elementary automata are "glued" by shared components of their state sets. Elementary automata can "communicate" by changing shared state components. Formally an APA is defined in the following way.

An *Asynchronous Product Automaton* consists of a family of *State Sets* $Z_S, S \in \mathbb{S}$, a family of *Elementary Automata* $(\Phi_e, \Delta_e), e \in \mathbb{E}$ and a *Neighbourhood Relation* $N : \mathbb{E} \rightarrow \mathcal{P}(\mathbb{S})$, where $\mathcal{P}(\mathbb{S})$ is the power set of X and $\mathbb{S}$ and $\mathbb{E}$ are index sets with the names of state components and elementary automata. For each Elementary Automaton (Φ_e, Δ_e)

— Φ_e is its *Alphabet* and

— $\Delta_e \subsetneq_{S \in N(e)}(Z_S) \times \Phi_e \times_{S \in N(e)}(Z_S)$ is its *State Transition Relation*

For each element of Φ_e the state transition relation Δ_e defines state transitions that change only the state components in $N(e)$.

An APA's (global) *States* are elements of$_{S \in \mathbb{S}}(Z_S)$. To avoid pathological cases it is generally assumed that $S = \cup_{e \in \mathbb{E}}(N(e))$ and $N(e) \neq \emptyset$ for all $e \in \mathbb{E}$. Each APA has one *Initial State* $s_0 = (q_{0S})_{s \in \mathbb{S}} \in$ $_{s \in \sim}(Z_S)$. In total, an APA $\mathbb{A}$ is defined by

$$\mathbb{A} = ((Z_S)_{s \in \mathbb{S}}, (\Phi_e, \Delta_e)_{e \in \mathbb{E}}, N, s_0)$$

The behavior of an APA is represented by all possible sequences of state transitions starting with initial state s_0. The sequence $(s_0, (e_1, a_1), s_1), (s_1, (e_2, a_2), s_2) \ldots$ with $a_i \in \Phi_{e_i}$ represents one possible sequence of actions of an APA. State transitions $(s_i, (e_{i+1}, a_{i+1}), s_{i+1})$ may be interpreted as labeled edges of a directed graph whose nodes are the states of an APA: $(s_i, (e, a), s_{i+1})$ is the edge leading from s_i to s_{i+1} and labeled by $(e; a)$. The subgraph reachable from the node s_0 is called the *reachability graph* of an APA.

There exists also the possibility to define subgroups of elementary automata which play a specific role in the whole APA. These subgroups share local state components and their common functionality can be determined by the state transitions assigned to the corresponding role.

2.3. Translating a WS-BPEL Process into APA

A WS-BPEL workflow definition introduces a model of Web Services interacting by exchanging sequences of messages between business partners. A BPEL process and its partners are defined as abstract WSDL services using abstract messages as defined by the WSDL model for message interaction.

To illustrate how APA can be used to model a WS-BPEL process we will begin with the translation of the partner links and the variables definitions. In the following definitions as identifiers of sets and variables we will use strings that are as close as possible to the corresponding attribute values used in the BPEL or WSDL definitions.

2.3.1. Data Type Definitions

`variable` definition Let consider the following definition of a WSDL `message`:

```
<message name = "mName">
    <part name = "pName1" type="pType1">
    <part name = "pName2" type="pType2">
<message/>
```

and a corresponding BPEL `variable` definition:

```
<variable name        = "vName"
          messageType = "mName"/>
```

Here we can assume that the types of the different WSDL part elements is *xsd:integer*, *xsd:string* or *xsd:boolean*. If the corresponding type is not one of these then it must be in a similar way previously defined as a structure of elements of simple types. The types *integer* and *boolean* are predefined in SHVT as **nat_0** and **bool**. If the value of the type attribute is *xsd:string*, then we have to define a set which contains all possible values as string constants:

$$pName1 = \{``t1_Value1", ``t1_Value2", \}$$
$$pName2 = \{``t2_Value1", ``t2_Value2", \}$$

For better readability and transparency of our definitions we specify one additional type, namely the value of the WSDL `message name` parameter:

$$mName_type = \{``mName"\}$$

Thus we can define a WSDL `message` as the cross product of the following sets:

$$mName = mName_type \times pName1 \times pName2.$$

If a `variable` attribute `messageType` has value $mName$ then it is uniquely determined what entries this variable must contain in order to be accepted as valid by the corresponding WS-BPEL process. According to the syntax, defined in SHVT, we get the following definitions of the above given sets

```
defset mName_type  = {'mName'};
defset pName1      = {'t1_Value1', 't1_Value2'};
defset pName2      = {'t2_Value1', 't2_Value2'};
defset mName       = pro(mName_type : gettype
                         pName1     : getpName1
                         pName2     : getpName2);
```

Definition 2.3.1.: `variable` data type

`partnerLink` definition Let consider the following WSDL `portType`, `partnerLinkType` and the BPEL `partnerLink` definitions:

```
<portType name = "ptName"
  <operation name = "opName">
    <input message = "mName1"/>
    <output message ="mName2"/>
    <fault name = "faultName"
           message = "mName3"/>
  </operation>
</portType>

<partnerLinkType name = "ltName">
  <role name = "roleName2">
    <portType name = "namePT1"/>
  </role>
```

```
  <role name = "roleName2">
    <portType name = "namePT2"/>
  </role>
</partnerLinkType>

<partnerLink myRole = "roleName1",
             name   = "plName",
             partnerLinkType = "ltName"/>
```

In the previous subsection we have described how we can translate a WSDL `message` into the APA syntax. Therefore, we can now assume that we have already defined the messages sets $mName1$, $mName2$ and $mName3$. First as a string constant we define the *operation* parameter:

$$ptName_operation_opName = \{'opName'\}$$

Then we define the *input*, *output* and *fault* types. Since they correspond to a given *operation* we represent them as a cross product of the following sets:

$$\begin{aligned} ptName_opName_output &= ptName_operation_opName \times mName2 \\ ptName_opName_input &= ptName_operation_opName \times mName1 \\ ptName_opName_fault &= ptName_operation_opName \times mName3 \end{aligned}$$

According to its WSDL definition each port is defined by the message type which is sent to or from this port.

$$\begin{aligned} ptName = \; & ptName_opName_input \; \cup \\ & ptName_opName_output \; \cup \\ & ptName_opName_fault \end{aligned}$$

A `partnerLinkType` characterizes the conversational relationship between two services by defining the “roles” played by each of the services in the conversation and specifying the `portType` provided by each service to receive messages within the context of the conversation. During such conversation it is very important for both sides to know exactly with which partner they are speaking to and the role of this partner during this conversation. For this purpose we specify a separate set $ServiceRefType$ with (String) constants, which will identify uniquely the each partner, supposed to take part in the whole process. If service plays only one role then we can define it with the following set:

$$ltName = ServiceRefType \times ServiceRefType \times ptName$$

The first parameter specifies the identifier of the service which sends a particular message, the second specifies the identifier of the service to which the message is sent and the third is the message itself. If a service, defined as a `partnerLinkType`, can play more than one role then it “listens” to more than one port. For each port we define a separate set

$$\begin{aligned} ptName1LT &= ServiceRefType \times ServiceRefType \times ptName1 \\ ptName2LT &= ServiceRefType \times ServiceRefType \times ptName2 \\ & \ldots \end{aligned}$$

Thus a `partnerLinkType` is exactly the union of all its “port” sets:

$$ltName = ptName1LT \cup ptName2LT \cup \ldots.$$

The definition of the `partnerLinkType` data type using the SHVT APA syntax is given in Definition 2.3.1..

```
defset ptName1_operation_opName = {'opName1'};
defset ptName1_opName_input  = pro(ptName_operation_opName,
                                    mName1);
defset ptName1_opName_output = pro(ptName_operation_opName,
                                    mName2);
defset ptName1_opName_fault  = pro(ptName_operation_opName,
                                    mName3)
defset ptName1    = ptName1_opName_input  ||
                    ptName1_opName_output ||
                    ptName1_opName_fault;
defset ptName1LT = pro(Partners,
                        Partners,
                        ptName1PT);

/* The second port type set "ptName2LT" is
   defined in the same way */
defset ltName = ptName1LT || ptName2LT;
```

Definition 2.3.1.: `partnerLinkType` data type

As we have already mentioned an APA can be seen as a family of elementary automata, "glued" together by shared components of their state sets and they can "communicate" by changing shared state components. In our model we consider the sequence of messages between the workflow and each of its partners as a state component. Since each `partnerLink` corresponds to a port type for the messages between the corresponding partner and the workflow, we interpret each `partnerLink` definition as a "container" for these messages, and thus as a global state component. According to the APA language and our interpretation, a WS-BPEL `partnerLink` definition with attribute `partnerLinkType = "ltName"` is given by

```
/* ltName_seq is a sequence of elements of type ltName */
def_state plName : ltName_seq = ::;
```

Definition 2.3.1.: `partnerLinkType` data type

2.3.2. *State Transitions and Roles in the APA Model*

Roles and state transitions. To each WSDL `partnerLinkType`, defined of the workflow, and the workflow itself we assign a separate role. Roles in SHVT are defined using the keyword **`def_role`**.

```
def_role roleName
{Local State Components}
{Macro Definitions};
```

Definition 2.3.2.: Role definition

Transitions and state patterns of a role are instantiated according to the set of symbols defined by using the keyword **def_pattern_bind**. This allows us to model workflows with multiple partners, assigned to one role (e.g. multiple client services which send requests to a workflow or different service partners which have to be contacted given that different events have occurred). The local state components, that can only be accessed by the automata of an instance of a partner, assigned to this role, are used to model the memory of that instance of a partner. All actions (elementary automaton) executed in a certain role are assigned to this role in the definition of a transition pattern.

As a single step, which changes the state of the APA, we consider the occurrence of any WS-BPEL activity (see Section *2.2.1.* or Section *2.2.1.*). When one of activities `receive`, `reply`, `invoke` occurs, then the input/output/error message is sent to the corresponding partner (i.e. elementary automaton) via one of the global state components.

Each single step is specified in a separate transition pattern assigned to one of the existing roles (`partnerLink`). The patterns specify conditions for state transitions and changes of the states of state components of the particular partner and of the shared state components. In order to define such patterns the function **def_trans_pattern** is used.

```
def_trans_pattern roleName pattern_label
(x1,x2,x3,...,xn)
allocations,
predicates,
actions;
```

Definition *2.3.2.*: Role definition

State transition pattern *pattern_label* will be created and assigned to role *roleName* followed by the used local variables. In the next lines allocations, predicates and actions can be specified. To perform state transition all predicates must be *true*.

Since each partner can send or recieve messages of a particular data type we have defined the global state components (the "containers" of such messages), so that they can store only messages of the corresponding type. Thus the only thing we have to do to assure that a state transition occurs is to check whether the corresponding global state component has stored a suitable (for the corresponding partner) message.

Initial state. To model a WS-BPEL process, it is necessary to give the initial state, of the APA model that is, the content of all state components in the state the process starts with. Since in the initial state of such process there are no messages sent or received by the partners we initialize the all global state components as the empty sequence (: :).

2.3.3. A WS-BPEL Workflow Example

Now we will describe in details with a concrete workflow example how in practice we model a WS-BPEL process. In the Appendix are given the WS-BPEL workflow definition, the WSDL and the APA description. The BPEL and WSDL code can be found also in [16].

The "Search Doctor" workflow implements the following scenario:

— The workflow is initiated when a message arrives requesting for a doctor to be assigned to a specific patient.

— A directory service is asked for the ID of the patient's doctor.

— The patient's doctor is contacted.

— If the patient's doctor replies positively, then the ID of the doctor is returned and the workflow terminates.

— If the patient's doctor replies negatively or does not reply at all within a timeframe then the search for a substitute doctor is initiated. In this case the call to the doctor is cancelled.

— A doctors' registry service is consulted for an appropriate substitute doctor.

— The suggested substitute doctor is contacted.

— If the doctor replies positively then the ID of the doctor is returned and the workflow terminates.

— If the doctor replies negatively or does not reply at all within a timeframe then the call is cancelled and steps 6-8 are repeated.

Figure *2.3.3.* shows the structure of an asynchronous product automaton modeling the above given scenario with one patient as a client. The circles represent state components and boxes are elementary

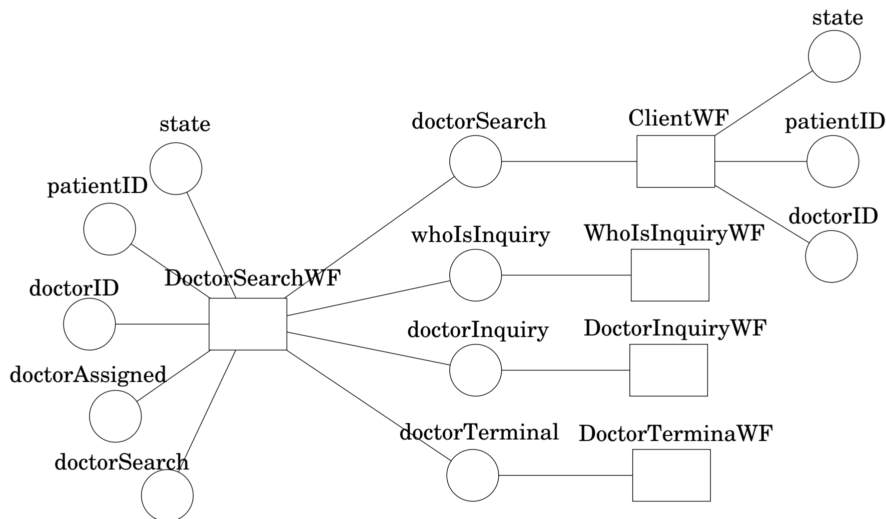

Figure 1: APA of the "Doctor Search" WS-BPEL workflow

automata. In the scenario take part 5 Partners: *DoctorSearchWF* (the workflow engine), *(*ClientWF) (the patient service), *DoctorTerminalWF* (the doctor service), *WhoIsInquiryWF* (the directory service) and *DoctorInquiryWF* (the doctors' registry service). Each partner taking part in the scenario is modeled by one elementary automaton that performs the partner's actions.

The state components *state*, *patientID*, *doctorID*, *doctorAssined* and *doctorSearch* of *DoctorSearchWF* service store the local state, the patient ID of the client service, the doctor ID of the found doctor (while no such is found this state component is empty), the information whether a suitable doctor has already been found or not (*true* or *false*) and the identifier of the instance of the client who has sent

the current request. The state components *state*, *patientID* and *doctorID* of *ClientWF* service store the local state, the patient ID of the patient, sending the request for a doctor search and the doctor ID is the local variable for storing the answer of the "Search Doctor" workflow.

The *doctorSearch* state component is used for communication between the workflow engine and the patient service, the *doctorInquiry* state component is used for communication between the workflow engine and the directory service, the *doctorTerminal* state component is used for communication between the workflow engine and the doctor service and the *whoIsInquiry* state component is used for communication between the workflow engine and the the doctors' registry service. These state components are shared between all partners (all elementary automata). A message is sent by adding it to the content of one of the described state component and received by removing it from the corresponding state component.

The neighbourhood relation (graphically represented by an arc) indicates which state components are included in the state of an elementary automaton and may be changed by a state transition of the elementary automaton. For example, automaton *DoctorSearchWF* may change its *state* and *doctorSearchWF* but cannot read or change the *state* of the automaton of *ClientWF*. The figure shows the structure of the automaton. The full specification of the automaton includes the state sets (the data types), transition relations of the elementary automata and the initial state. The full specification can be found in Section Appendix C. of the Appendix.

3. Security requirements for workflows

3.1. Introduction

In this section a framework for property-based characterisation of security requirements is revisited and appied to AmI environements with distributed workflows. The main goal is to provide a framework for the specification of a wide variety of workflow and services security requirements with formal semantics in terms of security properties of a discrete model of the workflow. Formalisations of authenticity, different types of non-repudiation and confidentiality and binding phases for workflows are presented within the framework.

A wide variety of approaches to security requirements specification has been developed. Among these one can distinguish two main approaches. In the first, a security model is used to express either an abstract view of a particular desired (secure) behaviour of a system or to specify undesired behaviour. In the second approach, formally characterised security properties are used to specify security requirements. One classical approach of the first category is the multi-level security model by Bell and La Padula [3]. The Bell-LaPadula model and other related security models, like the work by McLean [19], define models for access control and therefore do not meet all kind of security requirements of complex distributed workflows. The integrity model by Clark and Wilson [5] formulates restrictions on how data items might be altered. Other approaches are based on the specification of misuse cases describing threats by malicious agents or threat scenarios [17].

Concerning security properties, a large part of research work concentrates on information flow control and non-interference. The underlying trace based system models for some of these notions [24, 18] are closely related to the model presented in this paper. Non-interference and information flow properties can be used to describe various confidentiality properties. The underlying formal models are often highly complex and therefore accurate confidentiality requirements specification is error-prone and difficult. Another emphasis of research on security properties lies on authenticity and non-repudiation. Starting with the work on authentication logics [4], a wide variety of notions of authenticity has been published. Most of the formalisations are tailored for special cases like authentication protocol analysis while others provide more general definitions [22]. A wide variety of other fomalisations of security properties has been proposed, but there exists no framework in which different security requirements as required for workflows can be specified based on a single representation of a system. Furthermore, a lot of work exists on distributed workflows, but security issues are often neglected in this work.

The approach chosen in SERENITY for workflow security properties is based on a generic framework for security properties [8, 11, 10]. The underlying formal model describes system behaviours as (sets of) traces of actions, where these actions are associated with agents in the systems. This type of specification is very common, but for security properties additional information is required. First, satisfaction of security properties depends on the agents' view of the system. In this framework, this view has to be specified for each system as an alphabetic language homomorphism on traces of actions. Agents' views are used in other approaches as well (e.g. by Wedel and Kessler for the semantics of the authentication logic AUTLOG [23], or by Heisel et al. [14], defining the window to a system. However, agents' views in this approach are more flexible and thus, more adequate for distributed workflows in AmI environments. Second, for each agent the knowledge about the global workflow has to be part of the system specification. Obviously, satisfaction of confidentiality depends on the initial knowledge of the attacker. Furthermore, trust on underlying security mechanisms, such

as cryptographic algorithms for web-service security, are describe as knowledge about the system. The following sections describe and explain the model framework and its application to workflows.

3.2. System behaviour specification and agents' knowledge about a system

The *behaviour* S of a discrete system can be formally described by the set of its possible sequences of actions (traces). Therefore $S \subseteq \Sigma^*$ holds where Σ is the set of all actions of the system, and Σ^* is the set of all finite sequences of elements of Σ, including the empty sequence denoted by ε. This terminology originates from the theory of formal languages, where Σ is called the alphabet, the elements of Σ are called letters, the elements of Σ^* are referred to as words and the subsets of Σ^* as formal languages. Words can be composed: if u and v are words, then uv is also a word. This operation is called the *concatenation*; especially $\varepsilon u = u\varepsilon = u$. A word u is called a *prefix* of a word v if there is a word x such that $v = ux$. The set of all prefixes of a word u is denoted by pre(u); $\varepsilon \in$ pre(u) holds for every word u. The set of letters in a word u is denoted by $alph(u)$. In a distributed workflow actions can either represent actions controlling the workflow (e.g. calling a particular service or transferring the control over the workflow to another agent) or actions describing execution of a service.
Formal languages which describe system behaviour have the characteristic that pre$(u) \subseteq S$ holds for every word $u \in S$. Such languages are called *prefix closed*. System behaviour is thus described by prefix closed formal languages.
The set of all possible continuations of a word $u \in S$ is formally expressed by the *left quotient* $u^{-1}(S) = \{y \in \Sigma^* \mid uy \in S\}$.
Classical liveness and safety properties can easily be specified for such a system using well known formalisations. For security properties, the system model is extended by taking into account the agents' view of the system and agents' knowledge about the global system behaviour.

3.3. Agents' view and knowledge about the global system behaviour

Security properties can only be satisfied relative to particular sets of underlying system assumptions. Examples include assumptions on cryptographic algorithms (e.g. XML security mechanisms), secure storage (assumption on underlying devices providing a link to A3 security patterns), trust in the correct behaviour of agents or reliable data transfer. Relatively small changes in these assumptions can result in huge differences concerning satisfaction of security properties. Every model for secure systems must address these issues. Therefore, a model for secure systems needs to provide the means to accurately specify underlying system assumptions in a flexible way.
In order to provide the required flexibility, the system specification is extended by two components: *agents' knowledge* about the global system behaviour and *agents' view*. The knowledge about the system consists of all traces that an agent initially considers possible, i.e. all traces that do not violate any system assumptions, and the view of an agent specifies which parts of the system behaviour the agent can actually see. In the following paragraphs, these two components and their relations are explained in detail.
For any agent P its knowledge about the global system behaviour $W_P \subseteq \Sigma^*$ is considered to be part of the system specification.
We may assume for example that a message that was received by one agent in the distributed workflow must have been sent before. Thus an agent's W_P will contain only those sequences of actions in which a message is first sent and then received. All sequences of actions included in W_P in which a digital

signature is received and verified by using some agent Q's public key will contain an action where Q generated this signature.

Care must be taken when specifying the sets W_P for all agents P in order not to specify properties that are desirable but not guaranteed by verified system assumptions. E.g. digital singtures in XML data structures usually only contain parts of the data structure. Thus, one has to be careful to explicitly specify which parts of a message are actually protected by the signature.

The specification of the desired system behaviour generally does not include behaviour of malicious agents which has to be taken into account in open systems. An approach which is frequently used for the security analysis of cryptographic protocols is to extend the system specification by explicit specification of malicious behaviour. However, in general malicious behaviour is not previously known and one may not be able to adequately specify all possible actions of dishonest agents. In the framework used here, the explicit specification of agents' knowledge about system and environment allows to discard explicit specification of malicious behaviour. Every behaviour which is not explicitly excluded by some W_P is allowed. Denoting a system containing malicious behaviour by S and the correct system behaviour by S_C, we assume $S_C \subseteq S \subseteq \Sigma^*$. We further assume $S \subseteq W_P$, i.e. every agent considers the system behaviour to be possible. Security properties can now be defined relative to W_P. The relation between the system behaviour without malicious actions S_C, the system behaviour including malicious actions S, and W_P is graphically shown in Figure 2.

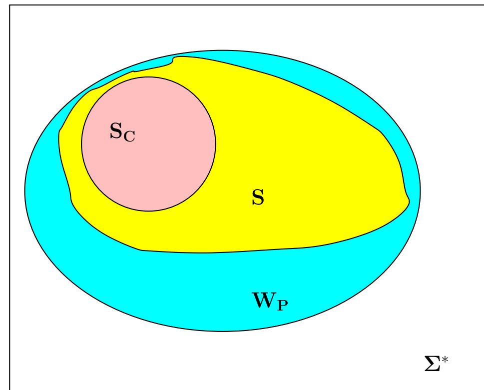

Figure 2: System behaviour and W_P

The set W_P describes what P knows initially. However, while a workflow is executed P usually learns from actions that have occurred. Satisfaction of security properties obviously also depends on what agents are able to learn. After a sequence of actions $\omega \in S$ has happened, every agent can use its *local view* of ω to determine the sequences of actions it considers to be possible. In order to determine what is the local view of an agent, we first assign every action to exactly one agent. Thus $\Sigma = \dot{\bigcup}_{P\in\mathbb{P}}\Sigma_{/P}$ (where $\Sigma_{/P}$ denotes all actions performed by agent P, and $\dot{\bigcup}$ denotes the disjoint union). The homomorphism $\pi_P : \Sigma^* \to {\Sigma^*}_{/P}$ defined by $\pi_P(x) = x$ if $x \in \Sigma_{/P}$ and $\pi_P(x) = \varepsilon$ if $x \in \Sigma \setminus \Sigma_{/P}$ formalizes the assignment of actions to agents and is called the $projection$ on P.

The projection π_P is the correct representation of P's view of the system if all information about an action $x \in \Sigma_{/P}$ is available for agent P and P can only see its own actions. In this case P's local view of the sequence of actions $\omega = (send, P, m1)(rec, Q, m1)$ for example is $(send, P, m1)$. However, P's view may be finer. For example it may additionally note other agents' actions without seeing the messages sent and received, respectively. In this case, P's local view of ω will be equal to $(send, P, m1)(rec, Q)$. P's local view may also be coarser than π_P. In a system the actions of which

are represented by a triple (*global state, transition label, global successor state*), although seeing its own actions, P will not be able to see the other agents' state.
Thus, the local view of an agent P on Σ is generally denoted by $\lambda_P : \Sigma^* \rightarrow \Sigma_P^*$. The local views of all agents together contain all information about the system behaviour S.
For a sequence of actions $\omega \in S$ and agent $P \in \mathbb{P}$, $\lambda_P^{-1}(\lambda_P(\omega)) \subseteq \Sigma^*$ is the set of all sequences that look exactly the same from P's local view after ω has happened. In the above example with the projection on P being P's local view, $\lambda_P^{-1}(\lambda_P(\omega))$ consists of sequences each of which contains an action $(send, P, (Q, m1))$. For some agent R that does not take part in ω, $\lambda_R^{-1}(\lambda_R(\omega))$ consists of sequences of actions of other agents, i.e. is equal to $(\Sigma \setminus \Sigma_{/R})^*$.
Depending on its knowledge about the system S, underlying security mechanisms and system assumptions, P does not consider all sequences in $\lambda_P^{-1}(\lambda_P(\omega))$ possible. Thus it can use its knowledge to reduce this set: $\lambda_P^{-1}(\lambda_P(\omega)) \cap W_P$ describes all sequences of actions P considers to be possible when ω has happened. The set $\lambda_P^{-1}(\lambda_P(\omega)) \cap W_P$ is similar to the possible worlds semantics that have been defined for authentication logics in the context of cryptographic protocols [1, 23]. The notion used here is more general because for authentication logics λ_P and W_P are fixed for all systems, whereas here they can be defined differently for different systems. The knowledge of P relative to a sequence of actions ω is graphically shown in Figure 3.

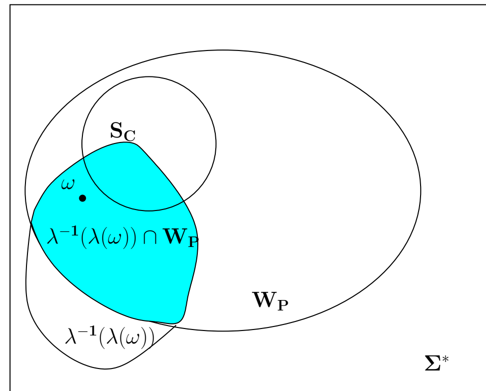

Figure 3: Local view of P

4. Examples for security requirements specification for workflows and services

In the following paragraphs we present and explain formal definitions for security properties, which are then illustrated using the examples of workflows and services.

4.1. Authenticity

4.1.1. Definition of the property

In the context of sequences of actions, authenticity is the authenticity of a particular action. Since usually authenticity of some action for a certain agent is required, the definition has to refer in some way to the agent. Thus we call a particular action $a \in \Sigma$ authentic for agent P if in all sequences that P considers possible after a sequence of actions ω has happened, some time in the past a must have happened. By extending this definition to a set of actions Γ being authentic for P if one of the actions in Γ is authentic for P we gain the flexibility that P does not necessarily need to know all parameters of the authentic action. For example, a message may consist of one part protected by a digital signature and another irrelevant part without protection. Then, the recipient can know that the signer has sent a message containing the signature, but the rest of the message is not authentic. Therefore, in this case, Γ comprises all messages containing the relevant signature and arbitrary other message parts.
The formal definition for authenticity is as follows.

Definition 1 (Authenticity) *A set of actions $\Gamma \subseteq \Sigma$ is authentic for $P \in \mathbb{P}$ after a sequence of actions $\omega \in S$ with respect to W_P if $alph(x) \cap \Gamma \neq \emptyset$ for all $x \in \lambda_P^{-1}(\lambda_P(\omega)) \cap W_P$.*

This definition and the following definition for proof of authenticity was presented in [8]. A similar definition of authenticity has been previously proposed by Schneider [22] who defines authenticity of events or sets of events on globally viewed traces.

4.1.2. Examples

In many applications based on Web services the authenticity of the result of a service is particularly important. I.e. the output of the Web service has originated from an authentic entity and it has not been modified since. In more detail, a typical Web service receives an input message and sends a reply. The reply is within a SOAP envelope. A specific part of the body of the SOAP message contains the actual reply, i.e. the info the requestor asked for. We want to be sure that the reply has originated from a specific entity (service provider) and that it has not been altered. In this case each time a partner in a workflow receives the result of a Web service the set Γ of actions that shall be authentic consists of all service executions by the Web service provider expected by the partner. Furthermore, the specification of the actions can also include time restrictions or bind the creation of the result to particular instances of the workflow.

Another example is concerned with access control, where a service provider requires the authenticity of a request by a partner in the workflow that is allowed to access the resources related to a service.

4.2. Proof of authenticity

4.2.1. Definition of the property

Some actions do not only require authenticity but also need to provide a proof of authenticity. If agent P owns a proof of authenticity for a set Γ of actions, it can send this proof to other agents, which in turn can receive the proof and be convinced of Γ's authenticity. In the following definition the set ΓP denotes actions that provide agents with proofs about the authenticity of Γ. If agent P has executed an action from ΓP then Γ is authentic for P and P can forward the proof to any other agent using actions in ΓS.

Definition 2 (Proof of authenticity) *A pair $(\Gamma S, \Gamma P)$ with $\Gamma S \subseteq \Sigma$ and $\Gamma P \subseteq \Sigma$ is a pair of sets of proof actions of authenticity for a set $\Gamma \subseteq \Sigma$ on S with respect to $(W_P)_{P \in \mathbb{P}}$ if for all $\omega \in S$ and for all $P \in \mathbb{P}$ with $alph(\pi_P(\omega)) \cap \Gamma P \neq \emptyset$ the following holds:*

1. *For P the set Γ is authentic after ω and*
2. *for each $R \in \mathbb{P}$ there exist actions $a \in \Sigma_{/P} \cap \Gamma S$ and $b \in \Sigma_{/R} \cap \Gamma P$ with $\omega ab \in S$.*

Agent $P \in \mathbb{P}$ can give proof of authenticity of $\Gamma \subseteq \Sigma$ after a sequence of actions $\omega \in S$ if 1 and 2 hold.

This definition represents one specific type of proofs. Kailar has classified proofs as strong or weak and transferable or non-transferable [15]. In terms of this classification our definition provides strong transferable proofs with the additional property that the possibility of the proof transfer is reliable. These proofs require the assumption that no agent disposes of its proofs. From a technical point of view the formal definition of this property is the most simple. However, other types of proofs can be formalized in a similar way. For example by introducing an additional class of actions representing the loss of proofs, Definition 2 can be modified: Agent P can give proof of authenticity only as long as no corresponding loss action has occurred.

4.2.2. Examples

Obviously, properties that can be realised by digital sinatures in the Web service request or response messages belong to this class of properties.
Another less obvious application of the notion of *proof of authenticity* is *auditability*. Meaning that there is a recording of the transactions of the Web service which ensures that both the ”requestor” and the service provider are accountable for their requests and responses, respectably. In this case, the writing and reading of the audit log constitutes the evidence.

4.3. A remark to integrity

In workflows, integrity is obviously a very important security requirement. However, consider the following example. Data integrity can be provided by a checksum. Change of the data would be detected because the checksum is not correct any more. But without authenticity, i.e. without knowing who computed the checksum, the checksum mechanism is useless: A malicious agent may change the data and simply compute a new checksum. Integrity is satisfied (meaning that data has not been changed since the checksum has been computed) but the actual security requirement is not.
Thus in order to make sense, integrity of data requires some kind of anchor fixing the origin of the data that shall remain unchanged. When considering distributed systems, this anchor can only be provided

by some authentic action in the past. Taking into account that integrity is implied by authenticity and that integrity without authenticity is valueless we have omitted a definition for integrity.

4.4. Confidentiality

In distributed workflows confidential data can occur in any service result. Partners in the workflow might have to transfer confidential data without getting any information about the content, while others are authorised to retrieve the data. Abviously, the definition of confidentiality requirements need to deal with information flow within the workflow and towards outsiders. Furthermore, the a suitable notion of confidentiality needs to be very flexible.

4.4.1. Definition of the property

Typically, the well-known concepts of non-interference or information flow control address confidentiality of actions: the occurrence or non-occurrence of certain actions of an agent shall not be deducible for another agent based on what it observes. In the literature there is a variety of formalizations of this concept, Mantel [18] gives a good insight into this topic. The subtle differences between these definitions show the spectrum of this kind of confidentiality.

However, non-interference is not suitable for the specification of security requirements in distributed open systems. Here, it can be assumed that all actions concerned with communication might indeed be visible to malicious agents. Nevertheless, confidentiality is required for data transmitted using these actions. An adequate notion of confidentiality therefore has to provide the flexibility to define confidentiality for arbitrary parameters of the actions. The notion of *parameter-confidentiality* presented in [9] provides this flexibility.

Various aspects are included in our definition. First, we have to consider agent C's view of the sequence ω it has monitored and thus the set of sequences $\lambda_C^{-1}(\lambda_C(\omega))$ that are, from C's view, identical. Second, C can discard some of the sequences from this set, depending on its knowledge of the system and the system assumptions, all formalized in W_C. There may for example exist interdependencies between parameters in different actions, such as a credit card number remaining the same for a long time, in which case C considers only those sequences of actions possible in which an agent always uses the same credit card number. In the simplified example in this paper, the password is never changed (in contrast to any good security practice). So the set of sequences C considers possible after having monitored ω is reduced to $\lambda_C^{-1}(\lambda_C(\omega)) \cap W_C$. Third, we need to identify the actions in which the respective parameter(s) shall be confidential. Usually many actions are independent from these and do not influence confidentiality, thus need not be considered.

Essentially, in our definition, parameter confidentiality is captured by requiring that for the actions that shall be confidential for agent C with respect to some parameter p, all possible (combinations of) values for p occur in this set. What are the possible combinations of parameters is the fourth aspect that needs to be specified, as we may want to allow C to know some of the interdependencies between parameters (e.g. C may be allowed to know that a message that was received must have been sent before).

Our definition of parameter confidentiality captures all these different aspects. However, formal details are not necessary for the specification of confidentiality requirements and are therefore omitted in this report. They can be found, along with the necessary explanations, in [9].
In order to specify parameter-confidentiality, the following information is needed:

α: the action(s) for which some parameter(s) shall be confidential,

$par(\alpha)$: a mapping par denoting the parameter(s) that shall be confidential

$\mathcal{P}$: the set of possible values for parameter $par(\alpha)$

$\mathbb{Q}$: a set of agents that is allowed to know the parameter value(s)

L: optional, a description of the interdependencies between parameter values.

4.5. Enforcing certain system behaviour

In particular in distributed workflows without a central workflow engine controlling the complete workflow, particular behaviour of parts of the workflow become security requirements. Certain behaviour of the global workflow shall not occur . Requiring authenticity of action a whenever action b has happened is one particular instantiation of enforcing system behaviour. However, other required behaviour needs a more general definition.

Definition 3 (enforce-behaviour) *For an alphabet Σ, let $L \subseteq \Sigma^*$ describe particular requirements and $S \subseteq \Sigma^*$ be the actual system.*
We say that L enforces its behaviour on S, denoted by enforce-behaviour(L,S)*, if $S \subseteq L$.*

This concept is particularly useful to define properties like the *transaction* property. A set of operations, which together form a transaction, are either all successfully performed or none is performed. As, for example, when we transfer money from one account to another. Two actions, withdrawal and deposit, form the ”money transfer” transaction. We can't have one without the other, so if deposit fails, then we have to ”undo” withdrawal, as well.

4.6. Refined workflow requirements

The previous chapter has introduced the basic notions that can be used for the accurate specification of S&D requirements and generic properties using these notions. However, the requirements directly described by notions are very general. Therefore, using them to specify more concrete S&D requirements as needed for SERENITY workflow pattern specifications can result in very complex expressions. Such expressions can be difficult to understand and the automatic processing of these expressions by the SERENITY Development-time Framework would require complex reasoning mechanisms. Consequently, refined notions have to be defined for workflows and services.

As a basis of these notions a concrete local view of agents and apropriate context descriptions fpr workflows are defined and then examples of requirements are given. More refined requirements can be easily added to the language based on these definitions.

— Agents' local view

The local view of agent P is restricted to those actions P performs itself and to actions **change-context** with the context that P can see:

$$\lambda_P(a) = \begin{cases} a & \text{if } a \in \Sigma_{/P} \\ \textit{change-context}(C_P) & \text{if } a = \textit{change-context}(C_{P_1}, \ldots, C_{P_k}) \text{ and } P \in \{P_1, \ldots, P_K\} \\ \varepsilon & \text{else} \end{cases}$$

The simplest case of a local view is that an agent can only see its own actions. However, in the context of AmI-based workflows where agents move and change the enviroments they are operating in, some actions, although not performed by the agent itself, will be noticeable. In the context of workflows and services such agents can be for example concerned with the set of possible services or service providers available to the agent. The action **change-context** can model such changes, hence can be seen by the agent if its own context is involved.

— Actions that can be learned from

As explained above, the homomorphism μ keeps those actions the agents can learn from, extracts the parameters that shall be confidential, and maps all other actions to the empty word. In the context of security properties on the level of workflows and services we have specifically considered confidentiality properties that refer to workflow data or to parameters of service requests or service results. We assume that in the model of a workflow, there are always actions *workflow-init(P,data)* and *workflow-result(P,results)* which describe that agent *P* initiates a workflow with data *data* and collects results. For the examples in this report we assume that all *data* and all *results* shall be confidential. In the case of service (e.g. web-service) requests we assume that there are actions *service-request(P,S,data* and *service-result(P,S,result)* for a service provided by *S* and used by *P*. Refined notions of confidentiality for subsets of data or agents involved can be easily constructed.

Thus we distinguish two different homomorphisms that extract workflow actions (μ_{wf}) and service actions (μ_{ws}).

$$\mu_{wf}(a) = \begin{cases} (workflow - init(P), data) & \text{if } a = workflow - init(P, data) \\ (workflow - result(P), results) & \text{if } a = workflow - result(P, results) \\ \varepsilon & \text{else} \end{cases}$$

$$\mu_{ws}(a) = \begin{cases} (service - request(P, S), data) & \text{if } a = service - request(P, S, data) \\ (service - result(P, S), result) & \text{if } a = service - result(P, S, result) \\ \varepsilon & \text{else} \end{cases}$$

For actions with different parameters the definitions have to be adapted accordingly.

— Dependencies between actions

The language $L \subseteq (\Sigma_t \times M)^*$ describes which dependencies between actions agents are allowed to know. In the context of workflows, very complex dependencies can occur and can be known to an attacker. For the examples in this report we assume that either no dependencies are known or that it is known that usually what is received must have been sent before. Thus, we consider two different languages L:

1. L_{Zero}: Agents are not allowed to know any dependencies.
2. $L_{SendRec}$ Agents are allowed to know that what is received must have been sent.

Based on these definitions, the following two sections introduce the refined notions of S&D requirements for workflows and services. In the following it is assumed that every action is associated with exactly one agent executing this action. Further it is assumed that some actions are characterized as *workflow-init*, *workflow-result*, *service-request*, *service-request-recieve*, *service-result*, and *service-result-receive*. These actions are used for managing workflows or using services such as web-services. Names are used for actions, parameters or agents in the descriptions as placeholders for arbitrary actions, parameters and agent names.

4.6.1. Requirements for single services

1. **authentic-service-request – Authenticity of service request for the server**
For a specific service request action **service-request** by client P and a specific receive action **service-request-recieve** by server S, **authentic(service-request,rec,P,S)** expresses the requirement that whenever S executes **service-request-receive(S,P,data)** the action **service-request(P,S,data)** is authentic for S corresponding to Definition 1. Note that this notion of authenticity (and also the following definitions) implies the integrity of *data*. Furthermore, it can be noted that integrity of *data* without authenticity of the service request is usually not useful, because instead of changing the parameters an attacker could simply impersonate P and issue a new service request action with different data.

2. **authentic-service-response – Authenticity of service result for the client**
For a specific service result action **service-result** by server S and a specific receive action **service-result-recieve** by client P, **authentic(service-result,rec,S,P)** expresses the requirement that whenever P executes **service-result-receive(P,S,data)** the action **service-request(S,P,data)** is authentic for P corresponding to Definition 1.

3. **authentic-local – Authenticity of local actions**
Such a property might be a useful requirement for server behaviour, e.g. in the context of stateful services or for the client requesting a service under particular conditions. For actions **local-action** and **confirm-action** both executed by the same agent A, **authentic-local(local-action,confirm-action,A)** expresses the requirement that whenever A executes **confirm-action** the action **local-action** is authentic for A corresponding to Definition 1.

4. **authentic-remote – Authenticity of remote actions**
Similar to the authenticity of local actions, internal remote actions might have to be authentic. One example could be the execution of some action on the server side prior to the action of sending the service result. For an action **remote-action** by agent B and an action **confirm-action** by agent A, **authentic-remote(remote-action,confirm-action,A)** expresses the requirement that whenever A executes **confirm-action** the action **remote-action** is authentic for A corresponding to Definition 1.

5. **non-rep-origin – Non-repudiation of service request**

   For a service request action **service-request** by client P and receive action **service-request-recieve** by server S, **non-rep-origin(service-request,P,S)** expresses the requirement that whenever S executes **service-request-recieve(S,P,data** the action **service-request(P,S,data)** is authentic for S and there exist proof actions by S for this action corresponding to Definition 2.

6. **non-rep-origin – Non-repudiation of service result**

   For a service request action **service-result** by server S and receive action **service-result-recieve** by client P, **non-rep-origin(service-result,S,P)** expresses the requirement that whenever P executes **service-result-recieve(P,S,data** the action **service-result(S,P,data)** is authentic for P and there exist proof actions by P for this action corresponding to Definition 2.

7. **non-rep-rec – Non-repudiation of receipt**
Similar to the definitions of non-repudiation of origin above, this non-repudiation of receipt

can also be applied to both, service request and service result. For a receive action **rec** by agent B and an action **confirm-rec** by agent A, **non-rep-rec(rec,confirm-rec,A)** expresses the requirement that whenever A executes **confirm-rec** the action **rec** is authentic for A and there exist proof actions by A for action **rec** corresponding to Definition 2.

8. In correspondance to authenticity requirements defined above, non-repudiation can also be defined for local and remote actions as **non-rep-local** and **non-rep-remote**.

9. **End-to-end confidentiality** This requirement is a requirement on data transfer: If data is confidential before the transfer it remains confidential (except for the recipient). No agents except those in **who** may learn the value of **data-to-be-conf** from the the actions in **actions-to-learn-from** although knowing the set **possible-values** that contains the possible values of the data to be confidential. In the context of services, end-to-end confidentiality is concerned with the communication between client and server for service requests and service results. The following more specific properties use the simpliest case where the only actions agents can learn from are service request and result send and receive actions. Other properties with more actions to learn from can be specified analogously. Further, we only consider the case were there are no dependencies between actions allowed to be known and the case where the only dependency that is allowed to be known is that before receiving a message it must have been sent.

   Hence the following definitions are based on the homomorphism μ_{com} and the languages L_{Zero} and $L_{SendRec}$ introduced above.

   Thus end-to-end-confidentiality has various different characteristics:

   — **end-to-end-conf-DepZero(action,data-to-be-conf,possible-values,who)**
   For all agents except those in **who**, the parameter **data-to-be-conf** in **action** shall be confidential. Agents only learn from send and receive actions (i.e. it is assumed that there are no hidden channels and dependencies to deduce the data) and are not allowed to know any dependencies between actions.

   — **end-to-end-conf-DepSendRec(action,data-to-be-conf, possible-values,who)**
   For all agents except those in **who**, the parameter **data-to-be-conf** in **action** shall be confidential. Agents only learn from send and receive actions and are allowed to know the dependencies between sending and receiving of messages.

10. **access-allow – Only agents explicitly allowed can get service results** We assume that for every service *WS* server S stores an access control list ACL_{WS}. For this requirement, service actions need to be refined in order to include the service as a parameter. Then *access-allow(WS,S,*$ACL_W S$*)* expresses the requirement that

    (a) **authentic(service-request,rec,P,S)**, i.e. all service requests by P are authentic,

    (b) **Authentic-local(service-request-recieve(S,P,WS,data),service-result(S,P,WS,result-data),S**, i.e. the server only sends results to those agents where a request was received,

    (c) Confidentialtity of the result-data, i.e. either

    (d) **end-to-end-conf-DepZero(result-data,possible-values,(P,S)** or **end-to-end-conf-DepSendRec(result-data, possible-values,(P,S))**

    (e) $\mathbf{P \in ACL_{WS}}$

This requirement can be refined for more complex access control policies.

11. **access-deny** Similar to the requirement **access-allow** a property can be defined that excludes particular agents from accessing a service.

4.6.2. Requirements for workflow processes

In contrast to the situation of requirements for single services defined in the previous section, workflows consist of sequences of service invocations. Thus, workflow requirements are mainly concerned with properties for all service calls or for particular phases during the workflow execution.

1. **authentic-service-request-all – Authenticity of all service requests for the server**
   For all service request actions **service-request** by client P and matching receive action **service-request-recieve** by server S, **authentic-all(service-request,rec,P,S)** expresses the requirement that whenever S executes a **service-request-receive(S,P,data)** the action **service-request(P,S,data)** is authentic for S corresponding to Definition 1.

2. **authentic-service-response-all – Authenticity of all service results for the client**
   For all service result actions **service-result** by server S and matching receive actions **service-result-recieve** by client P, **authentic-all(service-result,rec,S,P)** expresses the requirement that whenever P executes an action **service-result-receive(P,S,data)** the action **service-request(S,P,data)** is authentic for P corresponding to Definition 1.

3. **non-rep-origin-all – Non-repudiation of all service requests**
   For all service request action **service-request** by client P and matching receive action **service-request-recieve** by server S, **non-rep-origin(service-request,P,S)** shall hold.

4. **non-rep-origin-all – Non-repudiation of all service results**

   For all service request actions **service-result** by server S and receive actions **service-result-recieve** by client P, **non-rep-origin(service-result,S,P)** shall hold.

5. **non-rep-rec-all – Non-repudiation of receipt**
   Similar to the definitions of non-repudiation of origin for all actions above, this non-repudiation of receipt can also be applied to both, service request and service result.

6. The definition for confidentiality in the context of workflows is not different fro mthe definition for single services. The reason is, that a loss of confidentiality at any place within the workflow also violates the confidentiality requirements for single services. However, the validation of confidentiality properties within a complete worklow is considerably more complex than the validation for a single service invocation.

7. **Binding phases – enforcing particular workflow behaviours**
   Binding phases are a very powerful and flexible mechanism to specify workflow security requirements. The concept was introduced by Ochsenschläger and Grimm [7]. The main idea is, that usually in commerce processes, as well as in workflows, there are parts of the behaviour that can be arbitrarily ended. However, in some stages the process can reach a phase where only particular endings are desired. In this stage an attacker (and also a malicious participant in the workflow) could gain advantages by ending the workflow or continuing the workflow outside

the binding phase without passing through the specified *exit actions* for this phase. In other words, for a binding phase a set of valid endings of the phase have to be defined and all other exits to the phase have to be prevented. Formally, a binding phase can be defined by its **start action**, the set of all possible **end actions** and a predicate defining the phase.

Thus, the requirement **binding-phase(start-action,{exit-action1,exit-action2,...},phase-predicate** specifies the requirement that whenever action **start-action** occurs the predicate **phase-predicate** must be true until one of the exit actions in {**exit-action1,exit-action2,...**} occurs.

5. Conclusions

In this report, we have presented a framework for the specification of S&D Properties for workflows and services. We have followed a three steps approach. First, we have demonstrated the formal modelling of workflows in APA. We have provided precise guidelines for the translation of WS-BPEL specifications into APA. A tool that facilitates this translation is under development.

Second, we have provided formal definitions of basic S&D Properties for workflows and services. As these definitions are rather generic, as a third step, we formalised more concrete workflow and services properties, which constitute the primary elements of the *S&D Properties Specification Language for Workflows and Services*.

This language is flexible, extensible, and easy to use. It allows for the use of SHVT as a formal analysis tool, which has been enhanced to cover the scope of SERENITY and has been thoroughly used and tested in SERENITY. It is also inline with the work on the SERENITY Development-time Framework (SDF) and the corresponding S&D Classes and Patterns selection process.

Appendix A. BPEL

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- BPEL Process Definition -->
<process abstractProcess="yes" name="DoctorSearchWF"
targetNamespace="http://DoctorSearchWF"
xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
xmlns:ns1="http://workflow-security.org/wsdl/DoctorSearch/"
xmlns:wfpar="http://workflow-security.org/patterns/parameters/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <partnerLinks>
      <partnerLink myRole="doctorSearchService" name="doctorSearch"
partnerLinkType="ns1:doctorSearchLT"/>
      <partnerLink name="whoIsInquiry"
partnerLinkType="ns1:whoIsInquiryLT" partnerRole="whoIsService"/>
      <partnerLink name="doctorInquiry"
partnerLinkType="ns1:doctorInquiryLT" partnerRole="doctorRegistryService"/>
      <partnerLink myRole="waitingDoctorAnswer" name="doctorTerminal"
partnerLinkType="ns1:doctorTerminalLT" partnerRole="doctorAssignmentService"/>
      <partnerLink name="CancelDoctorCall"
partnerLinkType="ns1:CancelDoctorCallLT" partnerRole="cancelCallService"/>
   </partnerLinks>
   <variables>
      <variable messageType="ns1:patientIDMessage" name="patientID"/>
      <variable messageType="ns1:doctorIDMessage" name="doctorID"/>
      <variable messageType="ns1:doctorAssignedMessage"
name="doctorAssigned"/>
   </variables>
   <faultHandlers>
      <catch faultName="ns1:unableToFindDoctor">
         <reply faultName="ns1:unableToFindDoctor" name="ReplyWithFault"
operation="requestDoctor" partnerLink="doctorSearch"
portType="ns1:DoctorSearchServicePT"/>
      </catch>
      <catchAll>
         <reply faultName="ns1:unableToFindDoctor" name="ReplyGeneralFault"
operation="requestDoctor" partnerLink="doctorSearch"
portType="ns1:DoctorSearchServicePT"/>
      </catchAll>
   </faultHandlers>
   <eventHandlers>
      <onAlarm until="wfpar:generalAlarmTime">
         <throw faultName="ns1:unableToFindDoctor"/>
      </onAlarm>
   </eventHandlers>
   <sequence name="DoctorSearchSeq">
      <receive name="RequestForDoctor" operation="requestDoctor"
partnerLink="doctorSearch" portType="ns1:DoctorSearchServicePT"
variable="patientID"/>
      <invoke inputVariable="patientID" name="AskWhoIsPatientsDoctor"
operation="askWhoIsDoctor" outputVariable="doctorID" partnerLink="whoIsInquiry"
portType="ns1:WhoIsInquiryPT"/>
      <switch>
         <case condition="bpws:getVariableData('doctorID', 'doctorID') !
=&quot;&quot;">
```

```
            <sequence>
               <assign name="AssignEndpoint">
                  <copy>
                     <from part="endpointReference" variable="doctorID"/>
                     <to partnerLink="doctorTerminal"/>
                  </copy>
               </assign>
               <invoke inputVariable="patientID" name="CallDoctor"
operation="answerCall" partnerLink="doctorTerminal"
portType="ns1:DoctorListeningPT"/>
               <pick>
                  <onMessage operation="receiveAnswer"
partnerLink="doctorTerminal" portType="ns1:WaitingDoctorAnswerPT"
variable="doctorAssigned">
                     <!-- Process should terminate after this reply. -->
                     <reply name="ReplyPatientsDoctorID"
operation="requestDoctor" partnerLink="doctorSearch"
portType="ns1:DoctorSearchServicePT" variable="doctorID"/>
                   </onMessage>
                   <onAlarm for=" wfpar:callDoctorAlarmTime ">
                      <sequence>
                         <assign name="AssignFalseToDoctorAssigned">
                            <copy>
                               <from expression="false()"/>
                               <to part="assigned" variable="doctorAssigned"/>
                            </copy>
                         </assign>
                         <invoke inputVariable="patientID" name="CancelCall"
operation="cancelCall" partnerLink="CancelDoctorCall"
portType="ns1:CancelDoctorCallPT"/>
                      </sequence>
                   </onAlarm>
               </pick>
            </sequence>
         </case>
      </switch>
      <while condition="bpws:getVariableData('doctorAssigned', 'assigned') =
false()" name="WhileNoDoctorIsAssigned">
         <sequence>
            <invoke inputVariable="patientID" name="SuggestSubstituteDoctor"
operation="findSubstituteDoctor" outputVariable="doctorID"
partnerLink="doctorInquiry" portType="ns1:DoctorRegistryInquiryPT"/>
            <assign name="AssignEndpointSubDoctor">
               <copy>
                  <from part="endpointReference" variable="doctorID"/>
                  <to partnerLink="doctorTerminal"/>
               </copy>
            </assign>
            <invoke inputVariable="patientID" name="CallSubstituteDoctor"
operation="answerCall" partnerLink="doctorTerminal"
portType="ns1:DoctorListeningPT"/>
           <pick>
              <onMessage operation="receiveAnswer" partnerLink="doctorTerminal"
portType="ns1:WaitingDoctorAnswerPT" variable="doctorAssigned">
                 <reply name="ReplyDoctorID" operation="requestDoctor"
partnerLink="doctorSearch" portType="ns1:DoctorSearchServicePT"
```

```
variable="doctorID"/>
              </onMessage>
              <onAlarm until=" wfpar:callDoctorAlarmTime ">
                 <invoke inputVariable="patientID" name="CancelSubDocCall"
operation="cancelCall" partnerLink="CancelDoctorCall"
portType="ns1:CancelDoctorCallPT"/>
              </onAlarm>
           </pick>
         </sequence>
      </while>
   </sequence>
</process>
```

Appendix B. WSDL

```
<?xml version="1.0" encoding="UTF-8" ?>
<definitions
      targetNamespace="http://workflow-security.org/wsdl/DoctorSearch"
      xmlns="http://schemas.xmlsoap.org/wsdl/"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
      xmlns:bpel="urn:oasis:names:tc:wsbpel:2.0:process:executable"
      xmlns:tns="http://workflow-security.org/wsdl/DoctorSearch"
      xmlns:lns="http://workflow-security.org/wsdl/DoctorSearch">
<message name="patientIDMessage">
   <part name="patientID" type="xsd:string"/>
</message>

<message name="doctorIDMessage">
   <part name="doctorID" type="xsd:string"/>
   <part name="endpointReference" type="bpel:ServiceRefType"/>
</message>

<message name="errorMessage">
   <part name="errorCode" type="xsd:integer"/>
</message>

<message name="doctorAssignedMessage">
   <part name="assigned" type="xsd:boolean"/>
</message>

<!-- portTypes definitions -->
<portType name="DoctorSearchServicePT">
   <operation name="requestDoctor">
      <input message="lns:patientIDMessage"/>
      <output message="lns:doctorIDMessage"/>
      <fault name="unableToFindDoctor" message="lns:errorMessage"/>
   </operation>
</portType>

<portType name="WhoIsInquiryPT">
   <operation name="askWhoIsDoctor">
      <input message="lns:patientIDMessage"/>
      <output message="lns:doctorIDMessage"/>
      <fault name="noDoctorForPatient" message="lns:errorMessage"/>
   </operation>
</portType>

<portType name="WaitingDoctorAnswerPT">
   <operation name="receiveAnswer">
      <input message="lns:doctorAssignedMessage"/>
   </operation>
</portType>

<portType name="DoctorListeningPT">
   <operation name="answerCall">
   <input message="lns:patientIDMessage"/>
   </operation>
</portType>
```

```
<portType name="CancelDoctorCallPT">
   <operation name="cancelCall">
      <input message="lns:patientIDMessage"/>
   </operation>
</portType>

<portType name="DoctorRegistryInquiryPT">
   <operation name="findSubstituteDoctor">
      <input message="lns:patientIDMessage"/>
      <output message="lns:doctorIDMessage"/>
      <fault name="noDoctorAvailable" message="lns:errorMessage"/>
   </operation>
</portType>

<!-- partnerLinkType definitions -->
<plnk:partnerLinkType name="doctorSearchLT">
   <plnk:role name="doctorSearchService">
      <plnk:portType name="lns:DoctorSearchServicePT"/>
   </plnk:role>
</plnk:partnerLinkType>

<plnk:partnerLinkType name="whoIsInquiryLT">
   <plnk:role name="whoIsService">
      <plnk:portType name="lns:WhoIsInquiryPT"/>
   </plnk:role>
</plnk:partnerLinkType>

<plnk:partnerLinkType name="doctorTerminalLT">
   <plnk:role name="waitingDoctorAnswer">
      <plnk:portType name="lns:WaitingDoctorAnswerPT"/>
   </plnk:role>
   <plnk:role name="doctorAssignmentService">
     <plnk:portType name="lns:DoctorListeningPT"/>
   </plnk:role>
</plnk:partnerLinkType>

<plnk:partnerLinkType name="CancelDoctorCallLT">
   <plnk:role name="cancelCallService">
      <plnk:portType name="lns:CancelDoctorCallPT"/>
   </plnk:role>
</plnk:partnerLinkType>

<plnk:partnerLinkType name="doctorInquiryLT">
   <plnk:role name="doctorRegistryService">
      <plnk:portType name="lns:DoctorRegistryInquiryPT"/>
   </plnk:role>
</plnk:partnerLinkType>
</definitions>
```

Appendix C. APA Definition

```
/* ———— DEFINITIONS OF SETS ———— */
defset ServiceRefType = {'DoctorSearchWF',
                         'WhoIsInquiryWF',
                         'DoctorInquiryWF',
                         'ClientWF',
                         'DoctorTerminalWF',
                         'DoctorAnswerCallWF',
                         'DoctorP1',
                         'DoctorP2',
                         'Empty' };
defset ServiceRefType_seq = seq (ServiceRefType);

defset Event_Set = { 'callDoctorAlarmTime' };
defset Event     = pro (ServiceRefType, ServiceRefType, Event_Set);
defset Event_seq = seq(Event);

defset DoctorSearchWF_partner_links =
                   { doctorSearch, whoIsInquiry, doctorInquiry, doctorTerminal };

defset DoctorSearchWF_state_set = { 'init',
                                    'DoctorSearchSeq_1',
                                    'DoctorSearchSeq_2',
                                    'DoctorSearchSeq_3',
                                    'DoctorSearchSeq_4',
                                    'DoctorSearchSeq_5' };

defset ClientWF_state_set = { 'init', 'RequestSent', 'AnswerReceived' };

defset patientID              = { 'p1', 'p2', 'empty' };
defset patientIDMessage_type = { patientIDMessage };
defset patientIDMessage       = pro ( patientIDMessage_type ,
                                      patientID: patientID);
defset patientIDMessage_seq   = seq ( patientIDMessage );

defset doctorID              = { 'd1', 'd2', 'empty' };
defset doctorIDMessage_type = { doctorIDMessage };
defset doctorIDMessage       = pro ( doctorIDMessage_type,
                                     doctorID: doctorID,
                                     ServiceRefType : endpointReference);
defset doctorIDMessage_seq   = seq ( doctorIDMessage );

defset errorMessage_type = { errorMessage };
defset errorMessage      = pro ( errorMessage_type,
                                 nat_0 : errorCode );

defset doctorAssignedMessage_type = { 'doctorAssignedMessage' };
defset doctorAssignedMessage      = pro ( doctorAssignedMessage_type,
                                          bool: assigned );
defset doctorAssignedMessage_seq  = seq ( doctorAssignedMessage );

defset DoctorSearchServicePT_operation_requestDoctor = { 'requestDoctor' };
defset DoctorSearchServicePT_requestDoctor_input
       = pro ( DoctorSearchServicePT_operation_requestDoctor,
               patientIDMessage );
```

```
defset DoctorSearchServicePT_requestDoctor_output
       = pro ( DoctorSearchServicePT_operation_requestDoctor,
               doctorIDMessage );
defset DoctorSearchServicePT = DoctorSearchServicePT_requestDoctor_input ||
                               DoctorSearchServicePT_requestDoctor_output ||
                               errorMessage;
defset DoctorSearchLT          = pro ( ServiceRefType,
                                       ServiceRefType,
                                       DoctorSearchServicePT : message_DoctorSearchLT );

defset DoctorSearchLT_seq     = seq ( DoctorSearchLT );

defset WhoIsInquiryPT_operation_askWhoIsDoctor = { 'askWhoIsDoctor' };
defset WhoIsInquiryPT_askWhoIsDoctor_input
       = pro ( WhoIsInquiryPT_operation_askWhoIsDoctor,
               patientIDMessage );
defset WhoIsInquiryPT_askWhoIsDoctor_output
       = pro ( WhoIsInquiryPT_operation_askWhoIsDoctor,
               patientIDMessage,
               doctorIDMessage );
defset WhoIsInquiryPT      = WhoIsInquiryPT_askWhoIsDoctor_input ||
                             WhoIsInquiryPT_askWhoIsDoctor_output ||
                             errorMessage;
defset WhoIsInquiryLT      = pro ( ServiceRefType,
                                   ServiceRefType,
                                   WhoIsInquiryPT : message_WhoIsInquiryPT);
defset WhoIsInquiryLT_seq = seq ( WhoIsInquiryLT );

defset WaitingDoctorAnswerPT_operation_receiveAnswer = { 'receiveAnswer' };
defset WaitingDoctorAnswerPT_receiveAnswer_input
       = pro ( WaitingDoctorAnswerPT_operation_receiveAnswer,
               doctorAssignedMessage );
defset WaitingDoctorAnswerPT      = WaitingDoctorAnswerPT_receiveAnswer_input;
defset WaitingDoctorAnswerLT      = pro ( ServiceRefType,
                                          ServiceRefType,
                                          WaitingDoctorAnswerPT );
defset WaitingDoctorAnswerLT_seq = seq ( WaitingDoctorAnswerLT );

defset DoctorListeningPT_operation_answerCall = { 'answerCall' };
defset DoctorListeningPT_answerCall_input
       = pro ( DoctorListeningPT_operation_answerCall,
               patientIDMessage );
defset DoctorListeningPT      = DoctorListeningPT_answerCall_input;
defset DoctorListeningLT      = pro( ServiceRefType,
                                     ServiceRefType,
                                     DoctorListeningPT );
defset DoctorListeningLT_seq = seq ( DoctorListeningLT );

defset DoctorTerminalLT       = WaitingDoctorAnswerLT || DoctorListeningLT || Event;
defset DoctorTerminalLT_seq   = seq ( DoctorTerminalLT );

defset CancelDoctorCallPT_operation_cancelCall = { 'cancelCall' };
defset CancelDoctorCallPT_cancelCall_input
       = pro ( CancelDoctorCallPT_operation_cancelCall,
               patientIDMessage );
defset CancelDoctorCallPT      = CancelDoctorCallPT_cancelCall_input;
```

```
defset CancelDoctorCallLT     = pro ( ServiceRefType,
                                      ServiceRefType,
                                      CancelDoctorCallPT );
defset CancelDoctorCallLT_seq = seq (CancelDoctorCallLT);

defset DoctorRegistryInquiryPT_operation_findSubstituteDoctor
       = { 'findSubstituteDoctor' };
defset DoctorRegistryInquiryPT_findSubstituteDoctor_input
       = pro ( DoctorRegistryInquiryPT_operation_findSubstituteDoctor,
               patientIDMessage );
defset DoctorRegistryInquiryPT_findSubstituteDoctor_output
       = pro ( DoctorRegistryInquiryPT_operation_findSubstituteDoctor,
               doctorIDMessage  );
defset DoctorRegistryInquiryPT
       = DoctorRegistryInquiryPT_findSubstituteDoctor_input ||
         DoctorRegistryInquiryPT_findSubstituteDoctor_output ||
         errorMessage ;
defset DoctorRegistryInquiryLT
       = pro ( ServiceRefType,
               ServiceRefType,
               DoctorRegistryInquiryPT : message_DoctorRegistryInquiry);
defset DoctorRegistryInquiryLT_seq = seq ( DoctorRegistryInquiryLT );

/* ------ DEFINITIONS OF GLOBAL STATE COMPONENTS ------ */
def_state doctorSearch   : DoctorSearchLT_seq          := ::;
def_state whoIsInquiry   : WhoIsInquiryLT_seq          := ::;
def_state doctorTerminal : DoctorTerminalLT_seq        := ::;
def_state doctorInquiry  : DoctorRegistryInquiryLT_seq := ::;

/* ------- DEFINITIONS OF FUNCTIONS -------- */
defcase MainPatientDoctorDB : patientIDMessage >> doctorIDMessage
    MainPatientDoctorDB( patID ) = if( patientID( patID )='p1')
                                   then ('doctorIDMessage', 'd1', 'DoctorP1')
                                   else ('doctorIDMessage', 'd2', 'DoctorP2');

defcase SubstitutePatientDoctorDB : patientIDMessage >> doctorIDMessage
    SubstitutePatientDoctorDB( patID )
              = if( patientID( patID ) = 'p1' )
                    then ('doctorIDMessage', 'd2', 'DoctorP2')
                    else ('doctorIDMessage', 'd1', 'DoctorP1');

/* ------- DEFINITIONS OF ROLES AND TRANSITION PATTERRNS ------- */

/* ------- ROLE DoctorInquiryWF ------- */
def_role DoctorInquiryWF;

def_trans_pattern DoctorInquiryWF findSubstitute
    (from, patID)
        doctorInquiry ~= ::,
        message_DoctorRegistryInquiry( seg(doctorInquiry, 1, 1) ) ?
                        DoctorRegistryInquiryPT_findSubstituteDoctor_input,
        (from, 'DoctorInquiryWF', ('findSubstituteDoctor', patID)) << doctorInquiry,
        ('DoctorInquiryWF',
         from,
         ('findSubstituteDoctor', SubstitutePatientDoctorDB( patID ))
        ) >> doctorInquiry;
```

```
/* ———— ROLE WhoIsInquiryWF ————— */
def_role WhoIsInquiryWF;

def_trans_pattern WhoIsInquiryWF getDoctor
    ( from, patID )
         message_WhoIsInquiryPT( seg ( whoIsInquiry, 1, 1 ) ) ?
                        WhoIsInquiryPT_askWhoIsDoctor_input,
         (from, 'WhoIsInquiryWF' ,('askWhoIsDoctor', patID)) << whoIsInquiry,
         ('WhoIsInquiryWF',
          from,
          ('askWhoIsDoctor', patID, MainPatientDoctorDB( patID ))
         ) >> whoIsInquiry;

/* ———— ROLE DoctorTerminalWF ———— */
def_role DoctorTerminalWF;

def_trans_pattern DoctorTerminalWF AnswerCall
    (from, to, patID)
       doctorTerminal ~= ::,
       seg( doctorTerminal, 1, 1 ) ? DoctorListeningLT,
       (from, to,('answerCall',patID)) << doctorTerminal,
       (to,
        from,
        ('receiveAnswer',('doctorAssignedMessage', true))
       ) >> doctorTerminal;

/* ———— ROLE DoctorClientWF ———— */
def_role ClientWF
    { state : ClientWF_state_set :=  'init',
      local_patientID : patientIDMessage_seq := ('patientIDMessage','p1'),
      local_doctorID  : doctorIDMessage_seq  := ::
    };

def_trans_pattern ClientWF SendRequest
   ()
      doctorSearch = ::,
      ClientWF_state = 'init',
      ('ClientWF',
       'DoctorSearchWF',
       ('requestDoctor', ClientWF_local_patientID)
      ) >> doctorSearch,
      ClientWF_state := 'RequestSent';

def_trans_pattern ClientWF ReceiveAnswer
  (from, docID)
      doctorSearch ~= ::,
      message_DoctorSearchLT(seg ( doctorSearch, 1, 1 )) ?
                 DoctorSearchServicePT_requestDoctor_output,
      ClientWF_state = 'RequestSent',
      (from, 'ClientWF',('requestDoctor',docID)) << doctorSearch,
      ClientWF_local_doctorID := docID,
      ClientWF_state := 'init';

/* ———— ROLE DoctorSearchWF ———— */
def_role DoctorSearchWF
```

```
    { state : DoctorSearchWF_state_set := 'init',
      local_doctorSearch : ServiceRefType_seq := ::,
      patientID : patientIDMessage_seq := ::,
      doctorID : doctorIDMessage_seq := ::,
      doctorAssigned : doctorAssignedMessage_seq := ::
     };

def_trans_pattern DoctorSearchWF RequestForDoctor
   (from,ID)
         DoctorSearchWF_state = 'init',
         message_DoctorSearchLT(seg(doctorSearch, 1, 1)) ?
                    DoctorSearchServicePT_requestDoctor_input,
        (from, 'DoctorSearchWF', ('requestDoctor',ID)) << doctorSearch,
         DoctorSearchWF_patientID := ID,
         DoctorSearchWF_local_doctorSearch := from,
        ('DoctorSearchWF','WhoIsInquiryWF',('askWhoIsDoctor',ID)) >> whoIsInquiry,
         DoctorSearchWF_state := DoctorSearchSeq_1;


def_trans_pattern DoctorSearchWF AskWhoIsPatientsDoctor
   (patID, docID)
        DoctorSearchWF_state = DoctorSearchSeq_1,
        message_WhoIsInquiryPT( seg( whoIsInquiry, 1, 1 ) ) ?
                     WhoIsInquiryPT_askWhoIsDoctor_output,
        ('WhoIsInquiryWF',
         'DoctorSearchWF',
         ('askWhoIsDoctor',patID, docID)
        ) << whoIsInquiry,
        DoctorSearchWF_patientID := patID,
        DoctorSearchWF_doctorID := docID,
        when doctorID(docID)~='empty' {
           ('DoctorSearchWF',
            endpointReference(docID),
            ('answerCall',patID)
           ) >> doctorTerminal,
           DoctorSearchWF_state := DoctorSearchSeq_2
        };

def_trans_pattern DoctorSearchWF DoctorSearchSeq_pick_1
    (doctorAssigned)
        DoctorSearchWF_state = DoctorSearchSeq_2,
        seg(doctorTerminal, 1, 1) ? WaitingDoctorAnswerLT,
        (endpointReference( DoctorSearchWF_doctorID ),
         'DoctorSearchWF',
         ('receiveAnswer',doctorAssigned)
        ) << doctorTerminal,
        DoctorSearchWF_doctorAssigned := doctorAssigned,
        when assigned(doctorAssigned) = 'true' {
          ('DoctorSearchWF',
           'ClientWF',
           ('requestDoctor',DoctorSearchWF_doctorID)
          ) >> doctorSearch,
          DoctorSearchWF_state := init
        },
        when assigned(doctorAssigned) = 'false' {
          DoctorSearchWF_state := DoctorSearchSeq_3
```

```
      };

def_trans_pattern DoctorSearchWF DoctorSearchSeq_pick_2
    (from)
       DoctorSearchWF_state = DoctorSearchSeq_2,
       seg(doctorTerminal, 1, 1) ? Event,
       (from,
        'DoctorSearchWF',
        'callDoctorAlarmTime') << doctorTerminal,
       DoctorSearchWF_doctorAssigned := ('doctorAssignedMessage','false'),
       DoctorSearchWF_state := DoctorSearchSeq_3;


def_trans_pattern DoctorSearchWF DoctorSearchSeq_WhileNoDoctorIsAssigned_AskForSubstitute
    ()
       DoctorSearchWF_state = DoctorSearchSeq_3,

       if assigned(DoctorSearchWF_doctorAssigned) = 'false'  {
           ('DoctorSearchWF',
            'DoctorInquiryWF',
            ('findSubstituteDoctor',DoctorSearchWF_patientID)
           ) >> doctorInquiry,
           DoctorSearchWF_state := DoctorSearchSeq_4  }
       else {
           DoctorSearchWF_state := init };


def_trans_pattern DoctorSearchWF DoctorSearchSeq_WhileNoDoctorIsAssigned_RequestAnswer
    (docID)
       DoctorSearchWF_state = DoctorSearchSeq_4,
       message_DoctorRegistryInquiry(seg(doctorInquiry, 1, 1)) ?
                     DoctorRegistryInquiryPT_findSubstituteDoctor_output,
       ('DoctorInquiryWF',
        'DoctorSearchWF',
        ('findSubstituteDoctor',docID)
       ) << doctorInquiry,
       DoctorSearchWF_doctorID := docID,
       when doctorID(docID) ~= 'empty' {
         /* invoke_event(doctorTerminal, ('answerCall', patientID)), */
         ('DoctorSearchWF',
          endpointReference(docID),
          ('answerCall',DoctorSearchWF_patientID)
         ) >> doctorTerminal
       },
       DoctorSearchWF_state := DoctorSearchSeq_5;

def_trans_pattern DoctorSearchWF DoctorSearchSeq_WhileNoDoctorIsAssigned_pick_onMessage
    (assigned)
       DoctorSearchWF_state = DoctorSearchSeq_5,
       seg(doctorTerminal, 1, 1) ? WaitingDoctorAnswerLT,
       (endpointReference(DoctorSearchWF_doctorID),
        'DoctorSearchWF',
        ('receiveAnswer', assigned)
       ) << doctorTerminal,
       DoctorSearchWF_doctorAssigned := assigned,
       when(assigned(assigned) = 'true'){
```

```
          ('DoctorSearchWF',
           'ClientWF',
           ('requestDoctor',DoctorSearchWF_doctorID)
          ) >> doctorSearch
        },
        DoctorSearchWF_state := DoctorSearchSeq_3;

def_trans_pattern DoctorSearchWF DoctorSearchSeq_pick_onAlarm
    (from)
        DoctorSearchWF_state = DoctorSearchSeq_5,
        seg(doctorTerminal, 1, 1) ? Event,
        (from, 'DoctorSearchWF', 'callDoctorAlarmTime') << doctorTerminal,
        DoctorSearchWF_doctorAssigned := ('doctorAssignedMessage', 'false'),
        DoctorSearchWF_state := DoctorSearchSeq_3;
```

References

[1] M. Abadi and M.R Tuttle, 1991. A Semantics for a Logic of Authentication. In *Tenth Annual ACM Symposium on Principles of Distributed Computing, Montreal, Canada*, pages 201–216.

[2] Al. Alves, A. Arkin, Sid Askary, Ch. Barreto, Ben Bloch, F. Curbera, M. Ford, Y. Goland, N. Kartha, C. K. Liu, S. Thatte, P. Yendluri, Alex Yiu, Al. Guízar, R. Khalaf, D. König, M. Marin V. Mehta, and Danny van der Rijn, 2007. Web services business process execution language version 2.0.

[3] D. Bell and L. La Padula, 1975. Secure computer systems: Unified exposition and multics interpretation. Technical report MTR-2997, Mitre Cooperation.

[4] M. Burrows, M. Abadi, and R. Needham, 1990. A Logic of Authentication. *ACM Transactions on Computer Systems*, 8.

[5] D. Clark and D. Wilson, 1987. A comparison of commercial and military computer security policies. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[6] Fraunhofer Institute for Secure Information Technology SIT, Darmstadt, 2007. *Simple Homomorphism Verification Tool - Manual*.

[7] R. Grimm and P. Ochsenschläger, 2001. Binding Cooperation, A Formal Model for Electronic Commerce. *Computer Networks*, 37:171–193.

[8] S. Gürgens, P. Ochsenschläger, and C. Rudolph, 2002. Authenticity and Provability – a Formal Framework. In *Infrastructure Security Conference InfraSec 2002*, volume 2437 of *Lecture Notes in Computer Science*, pages 227–245. Springer Verlag.

[9] S. Gürgens, P. Ochsenschläger, and C. Rudolph, 2003. Parameter confidentiality. In *Informatik 2003 - Teiltagung Sicherheit*. Gesellschaft für Informatik.

[10] S. Gürgens, P. Ochsenschläger, and C. Rudolph, 2005. Abstractions preserving parameter confidentiality. In *European Symposium On Research in Computer Security (ESORICS 2005)*, pages 418–437.

[11] S. Gürgens, P. Ochsenschläger, and C. Rudolph, 2005. On a formal framework for security properties. *International Computer Standards & Interface Journal (CSI), Special issue on formal methods, techniques and tools for secure and reliable applications*.

[12] S. Gürgens and C. Rudolph, 2004. Security Analysis of (Un-) Fair Non-repudiation Protocols. *Formal aspects of computing*.

[13] S. Gürgens, C. Rudolph, D. Scheuermann, M. Atts, and R. Plaga, 2007. Security evaluation of scenarios based on the TCG's TPM specification. In Joachim Biskup and Javier Lopez, editors, *Computer Security - ESORICS 2007*, volume 4734 of *Lecture Notes in Computer Science*. Springer Verlag.

[14] M. Heisel, A. Pfitzmann, and T. Santen, 2001. Confidentiality-preserving refinement. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, pages 295–305. IEEE Computer Society Press.

[15] R. Kailar, 1996. Accountability in Electronic Commerce Protocols. *IEEE Transactions on Software Engineering*, 22(5):313–328.

[16] S. Kokolakis, P. Kostaki, P. El Khoury, and C. Pandolfo, 2006. A2.D4.1-Initial Set of S§D Patterns for Workflows. Technical report.

[17] V. Lotz, 1997. Threat Scenarios as a Means to Formally Develop Secure Systems. *Journal of Computer Security 5*, pages 31 – 67.

[18] H. Mantel, 2000. Possibilistic definitions of security – an assembly kit. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 185–199.

[19] J. McLean, 1990. The specification and modeling of computer security. *IEEE Computer*, 23(1):9–16.

[20] P. Ochsenschläger, J. Repp, and R. Rieke, 2000. Abstraction and composition – a verification method for co-operating systems. *Journal of Experimental and Theoretical Artificial Intelligence*, 12:447–459. Copyright: ©2000, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

[21] P. Ochsenschläger, J. Repp, R. Rieke, and U. Nitsche, 1999. The SH-Verification Tool – Abstraction-Based Verification of Co-operating Systems. *Formal Aspects of Computing, The Int. Journal of Formal Methods*, 11:1–24.

[22] S. Schneider, 1996. Security Properties and CSP. In *Symposion on Security and Privacy*. IEEE.

[23] G. Wedel and V. Kessler, 1996. Formal Semantics for Authentication Logics. In *Computer Security - Esorics 96*, volume 1146 of *LNCS*, pages 219–241.

[24] A. Zakinthinos and E. Lee, 1997. A general theory of security properties. In *Proceedings of the IEEE Symposium on Security and Privacy*.