MAnagement of Security information and events
in Service InFrastructures

MASSIF
FP7-257475

D4.2.2 - Process Model and Dynamic Simulation and Analysis Modelling Framework

Activity	A4	Workpackage	WP4.2
Due Date	Month 24	Submission Date	2012-10-15
Main Author(s)	Roland Rieke (Fraunhofer) Jürgen Repp, Maria Zhdanova (Fraunhofer)		
Version	v1.0	Status	Final
Dissemination Level	PU	Nature	R
Keywords	predictive security analysis, analysis of process behavior, security modeling and simulation, security monitoring		
Reviewers	Herve Debar (IT) Rodrigo Diaz (Atos)		

Part of the Seventh
Framework Programme
Funded by the EC - DG INFSO

Version history

Rev	Date	Author	Comments
V0.1	2012-06-04	Roland Rieke (SIT)	initial version
V0.2	2012-06-17	Roland Rieke (SIT)	section guidelines introduced
V0.9	2012-10-05	Roland Rieke, Juergen Repp (SIT)	version for review
V1.0	2011-10-15	Elsa Prieto (Atos)	final review and official delivery

Glossary of Acronyms

AMSEC	Attack Modeling and Security Evaluation Component
APA	Asynchronous Product Automata
BAM	Business Activity Monitoring
BPEL	Business Process Execution Language
CEP	Complex Event Processing
EPC	Event-driven Process Chain
ERP	Enterprise Resource Planning
GET	Generic Event Translation
GUI	Graphical User Interface
IDMEF	The Intrusion Detection Message Exchange Format
ISO 8601	ISO (International Organization for Standardization) Representations of dates and times, 1988-06-15
GMT	Greenwich Mean Time
LTL	Linear Temporal Logic
LTS	Labeled Transition System
NIST	National Institute of Standard and Technologies
NTP	Network Time Protocol
PL	Preamble Language
PM	Process Modeller
PN	Product Net
PNML	Petri Net Markup Language
PSA	Predictive Security Analyser
RG	Reachability Graph
SEM	Security Event Modeller
SHVT	Simple Homomorphism Verification Tool
SIEM	Security Information and Event Management
WFM	Workflow Management

WS-BPEL	Web Services Business Process Execution Language
WSDL	Web Services Description Language
XES	Extensible Event Stream
XML	Extensible Markup Language
XSD	XML Schema Definition

Executive Summary

Security analysis is growing in complexity with the increase in functionality, connectivity, and dynamics of current electronic business processes. To tackle this complexity, the application of models is becoming standard practice. Considering today's frequent changes to business process models and instances, model-based support for security analysis is not only needed in pre-operational phases but also at runtime. In this document, we present an approach to support model-based evaluation of the current security status of business process instances as well as to allow for decision support by analysing close-future process states. Our approach is based on operational formal models derived from development-time process and security models. This document exemplifies our approach utilising processes from the MASSIF scenarios and demonstrates the systematic development and application of models for security analysis at runtime.

To support such security analysis at runtime, we utilise formal models derived on informal or semi-formal process and security models. The formalised process model subscribes to events from the MASSIF complex event processing system, consumes them via the repository and reflects the current process state. A security analysis model identifies current and close-future violations of the security policy. A mapping model maps transitions in the security analysis model onto security alerts that are provided via the repository to the decision support and reaction component and the MASSIF visualisation service.

Contents

List of Figures

List of Tables

Listings

1 Introduction

The overall objective of the work presented here is to develop advanced techniques for the evaluation of security-related events and their interpretation with respect to the known control-flow of the processes involved and the required security properties. These techniques should enable methodologies for performing dynamic predictive process analysis at runtime, detecting any prospective potential violation of the required security properties. Finally the developed methods should be implemented into a prototype, featuring a new generation, intelligent, multi-domain security event-processing and predictive security monitoring and simulation, which will be delivered in D4.2.3.

1.1 Purpose & Scope

This deliverable is part of work package WP4.2. The particular objectives of this work package are

- to specify an executable event-driven process model triggered by real-time events,
- to develop methodologies for performing dynamic predictive process analysis at runtime,
- to provide techniques, featuring the ability to perform intensive simulation analysis under given hypothesis, and
- to implement the provided techniques in an intelligent security event-processing engine.

The identification, correlation and aggregation of events is a core part in the detection of current security threats during runtime. However, depending on the current security level or the process specification itself, security-related events can have different effect on the security of different processes. Therefore, in order to be able to interpret abstract security-related events in the context of specific security properties and processes, new techniques have been developed, which allow the analysing tool to perform predictive process analysis at runtime.

This deliverable provides the results of the Tasks T4.2.2 “Process Model” and T4.2.3 “Dynamic Simulation and Analysis Methods”. This comprises a description of a formal security-aware process model; specification of techniques for the identification of close-future security-threatening process states and simulation analysis methods, which inspect the behaviour of complex/parallel processes under given hypotheses. The operational process model is supplemented by a model of the required security properties that the corresponding process should fulfil. This security model is represented in form of a monitor automaton.

Examples based on informal process specifications from the scenarios in [7], semi-formal process descriptions provided in Event-driven Process Chain (EPC) syntax by Atos, and processes discovered by external tools and provided in Petri net syntax served as input for the development and test of the implemented algorithms. The deliverable describes how the informal and semi-formal process models can be converted to a formal event-driven process model. On the base of this model it is possible to use different analysis and simulation techniques, which enable the evaluation of the security status during runtime, based on realtime events. Further, it can detect potential violations of the security requirements and generate respective alarms. Moreover, it allows to inspect possible effects of changes in process specifications or security requirements by performing an intensive simulation analysis.

The developed methods and algorithms provide the basis of the dynamic simulation and monitoring tool which will be implemented in the task T4.2.4.

1.2 Related Work

The work presented here combines specific aspects of security analysis with generic aspects of process monitoring, simulation, and analysis. The background of those aspects is given by the utilization of models at runtime [14]. A blueprint for our architecture of predictive security analysis is given in [34].

A formalised approach for security risk modelling in the context of electronic business processes is given in [42]. It touches also the aspect of simulation, but does not incorporate the utilization of runtime models. A refinement methodology and modelling language for the purpose of developing secure electronic business processes based on early requirements analysis is proposed by [38]. The proposal touches runtime enforcement of security mechanisms but does not allow for an evaluation of the security status of business processes at runtime. A model-driven approach focusing on access control for business process models is provided in [49]. It allows for the annotation of security goals to business process models, the validation of annotated process models using model checking and the generation of configuration artifacts for runtime components. Runtime analysis of security properties is not addressed by this approach. Further current approaches for the analysis and specification of security properties of business process models at development-time are given by [3, 4, 48, 15].

Approaches that focus security models at runtime are given in [27, 25]. Morin et. al propose a novel methodology to synchronise an architectural model reflecting access control policies with the running system [27]. Therefore, the methodology emphasises policy enforcement rather than security analysis. The integration of runtime and development-time information on the basis of an ontology to engineer industrial automation systems is discussed in [25].

Process monitoring has gained some popularity recently in the industrial context prominently accompanied with the term Business Activity Monitoring (BAM). The goal of BAM applications, as defined by Gartner Inc., is to process events, which are generated from multiple application systems, enterprise service buses, or other inter-enterprise sources in real-time in order to identify critical business key performance indicators, get a better insight into the business activities, and thereby improve the effectiveness of business operations [24]. Recently, runtime monitoring of concurrent distributed systems based on Linear Temporal Logic (LTL), state-charts, and related formalisms has also received some attention [18, 23]. However, these works are mainly focused on error detection, e.g., concurrency related bugs. A classification for runtime monitoring of software faults is given in [8]. Schneider [37] analysed a class of safety properties and related enforcement mechanisms that work by monitoring execution steps of some target system, and terminating the target's execution, whenever it is about to execute an operation, which would violate the security policy. Extensions of this approach are discussed in [5]. Patterns and methods to allow for monitoring security properties are developed in [39, 40, 43, 13]. In the context of BAM applications, in addition to these features we propose a *close-future* security analysis as it is detailed in Section 4.2.1. Our analysis provides information about possible security policy violations reinforcing the security-related decision support system components.

Different categories of tools applicable for simulation of business processes including process modelling tools are based on different semi-formal or formal methods such as Petri Nets [10] or EPC [9, 26, 45]. Some process management tools such as FileNet [28] offer a simulation tool to support the design phase. Also, some general-purpose simulation tools such as CPNTools [36] were proven to be suitable for simulating business processes. The process mining framework ProM [22] supports plug-ins for different types of models and process mining techniques. However, independently from the tools and methods used, such simulation tools concentrate on statistical aspects, redesign, and commercial optimisation of the business process. On the contrary, we propose an approach for *on-the-fly* dynamic simulation and analysis on the basis of operational Asynchronous Product Automata (APA) models (cf. Chapter 4). This includes consideration of the current process state and the event information combined with the corresponding steps in the process model. We consider the framework presented in [22] on run-time compliance verification for business processes as complementary to our work. In [34], we proposed to use APA to specify meta-events, which match security critical situations, to generate alerts. However, since this is slow and not easily usable by end-users, we decided to build the matching algorithm directly into the Predictive Security Analyser (PSA). We use monitor automata to specify the operational security requirements graphically (cf. Section 4.5.1).

1.3 Analysis of Security Requirements

According to requirements analysis and guidelines in Deliverable D.2.1.1 [7], besides issues like dependability, redundancy and fault tolerance, the analysis of the four scenarios considered reveals the need for enhanced *security-related* features of future Security Information and Event Management (SIEM) platforms. These features go beyond what is currently supported by existing solutions. Overall a lack of capability to model incidents at an abstract level is perceived. From the scenarios investigated, and the current SIEM limitations observed, the guideline in Table 1.1 has been identified to be relevant for the work within this deliverable.

Guideline	Description
G.S.4. Predictive security monitoring	Predictive security monitoring allows to counter negative future actions, proactively. There is a crucial demand for early warning capabilities. Moreover, the limitations with regards to the Managed Enterprise Service point to the fact that dealing with unknown or unpredictable behaviour patterns is not sufficient in current SIEM solutions.

Table 1.1: Guidelines concerning security addressed by this deliverable

In terms of *trustworthiness* considerations, we assume that data we use for analysis are already processed and provided by trustworthy MASSIF components.

In terms of *legal* considerations, we assume that data we use for analysis are already processed and provided in a form compliant with the legal requirements stated in [7]. With respect to the specific guideline *G.L.5* “Least Persistence Principle. Only data strictly needed for security guarantee must be kept, while unnecessary details must be deleted or made anonymous.”, we provide technical means, which can be applied to fulfil the related *Requirement 5.1* “Mechanisms must be provided to filter data containing information not relevant to security processing.". For this purpose, we provide a schema mapping, whereby all data containing information not relevant to security processing can be filtered out (cf. Section 4.3.2).

1.4 Glossary adopted in this deliverable

As agreed by the MASSIF Consortium, the main reference of security glossary is provided by the National Institute of Standard and Technologies (NIST) [20].

1.5 Structure of the document

The remainder of our paper is organised as follows. Chapter 2 describes our process modelling approach, an appropriate formal representation and the tool support.

Chapter 3 describes the conversion of process specifications into the formal representation, gives examples chosen from the scenarios described in deliverable D2.1.1 [7], and describes process discovery methods and tools as well as the transfer of the discovered process descriptions into the PSA. Chapter 4 describes the dynamic simulation and prediction methods as well as the monitoring of security requirements. Chapter 5 exemplifies our approach with two different processes from the MASSIF scenarios. Concluding remarks and an outlook to the further work in upcoming deliverables are given in Chapter 6.

2 Process Models for Security Analysis

In order to support security analysis at runtime, we utilise formal models based on development-time process, and security models. On the basis of sound methods for the elicitation and modelling of security requirements provided in [17, 32] and an architectural blueprint described in [34], we document in this deliverable our approach to analyse the security status of electronic business processes. A first sketch of this approach has already been published in [11]. This deliverable substantially extends the descriptions given before and includes major enhancements detailing the monitoring formalism, implementation, evaluation, and context of our approach. In particular, it provides the base for the PSA, which is subject of deliverable D4.2.3. The PSA prototype, which is currently in development in D4.2.3 will provide advanced, application aware security monitoring capabilities to the MASSIF SIEM. Specifically, it will support *close-future process behaviour simulation* and *prediction of possible security violations*.

This chapter starts with an introduction to informal, semi-formal, and formal process models in Section 2.1. In Section 2.2 we describe the formal basis for the operational models which we use in order to derive the prediction of the behaviour of the analysed processes in the near future. Finally, in Section 2.3 we describe the Process Modeller (PM), which currently comprises a textual Preamble Language (PL) editor, and a graphical Product Net (PN) editor.

2.1 Informal, Semiformal, and Formal Process Models

2.1.1 Introduction to EPC

Today, business process engineering, deployment and runtime control is supported by diverse tools, including Enterprise Resource Planning (ERP) and Workflow Management (WFM) tools. Some of the leading products, such as SAP R/3 [1] and ARIS [2], use EPCs to model business processes [45].

The EPC notation is a specific language, which is used to represent business processes graphically in form of a flowchart. An EPC graph shows the control flow structure of a process as a chain of events and functions, i.e., an event-driven process chain.

The EPC notation was first introduced by Keller, Nüttgens and Scheer in 1992 [19]. In the original publication [19] the basic constructs of an EPC process model are *functions* (sometimes also called *activities* or *actions*) and *events*. Functions represent active components, i.e., activities, tasks or process steps in a process, which are triggered by events. Events are passive, they represent the occurrence of a state which describes the situation before or after a function is executed. Logical and, or, and xor (exclusive or) operators are used to connect the basic constructs, in this way the flow of control is

[1] http://en.wikipedia.org/wiki/SAP_R/3

[2] http://en.wikipedia.org/wiki/Architecture_of_Integrated_Information_Systems

specified.

There is no consistent notation of basic constructs and operators used in the different publications and tools. The original publication [19] used rectangles to represent events, rounded rectangles to represent functions and circles with different inscriptions for the logical operators.

In this deliverable we use rectangles with rounded corners to denote EPC functions and *hexagons* to denote EPC events. There are many extensions to the original EPC notation available (cf. Figure 2.1). However for our purposes the basic constructs and operators are sufficient.

The EPC language is targeted to be easy to understand and use by people which understand the processes on the level of their business logic. These people are normally not familiar with formal specification methods. Therefore, a schema to transform an EPC model into an operational formal model was developed (cf. Section 2.2).

2.1.2 BPEL

Another frequently used notation with similar purpose as EPC is Business Process Execution Language (BPEL). A Web Services Business Process Execution Language (WS-BPEL) workflow definition introduces a model of Web Services interacting by exchanging sequences of messages between business partners. A WS-BPEL process and its partners are defined as abstract Web Services Description Language (WSDL) services using abstract messages as defined by the WSDL model for message interaction. WSDL is an Extensible Markup Language (XML)-based language that can be used to describe the functionality offered by a Web service.

BPEL is not used in the scenarios of MASSIF. However, in Section 3.2 we describe a transformation of BPEL processes specifications into an operational formal model for use with the PSA.

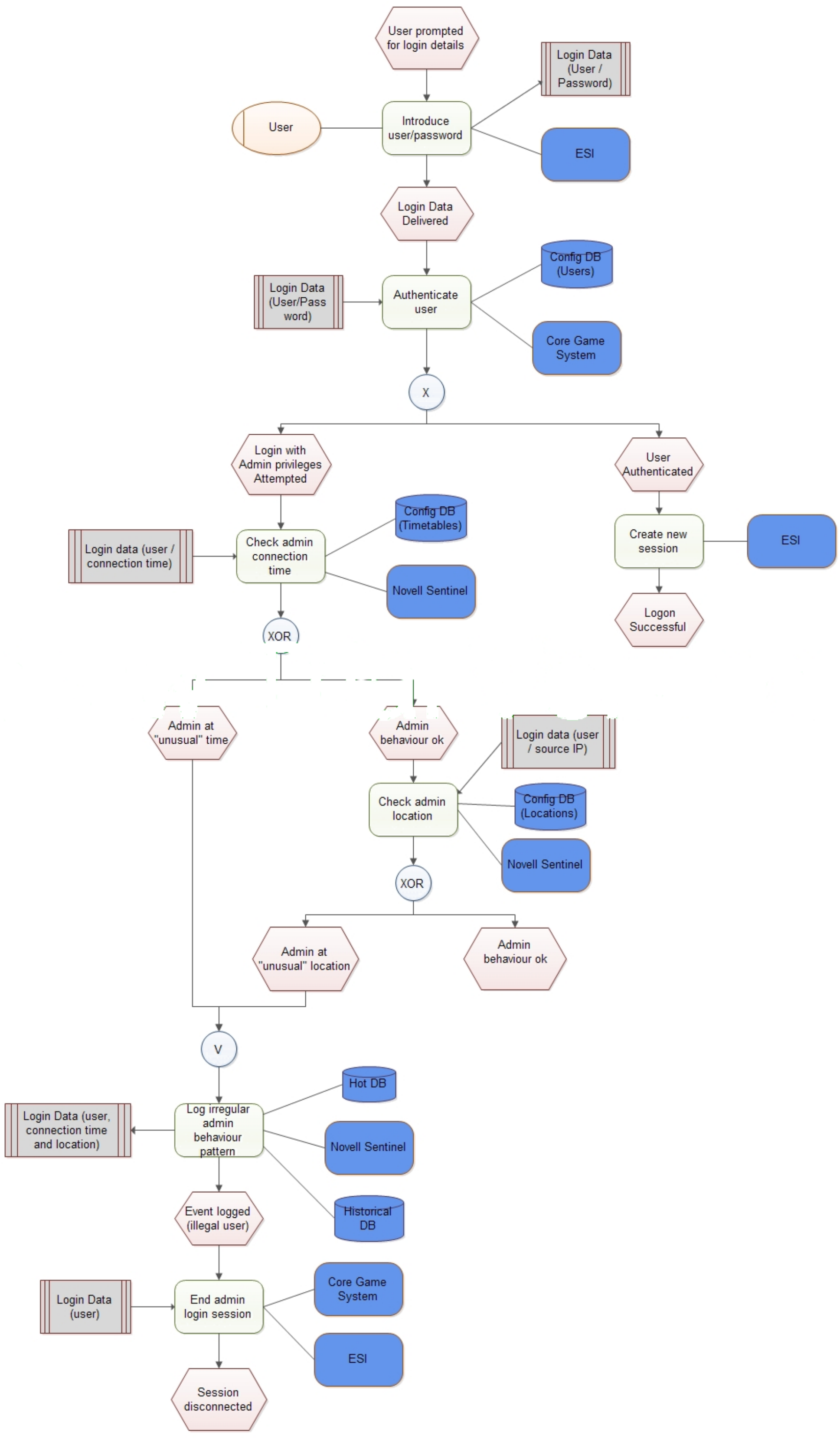


Figure 2.1: EPC Olympic Games

2.2 Formal Basis for Operational Process Models

We now introduce *formalised process models*, which will be utilised to reflect the current state of the system. Furthermore, the formalised process models will be the basis for the prediction of close-future actions.

In order to analyse the system behavior with tool support, an appropriate formal representation has to be chosen. In our approach, we use an operational finite state model of the behavior of the given process. This can be based on APA, a flexible operational specification concept for cooperating systems [30], or on PN, which are based on Petri nets [31] augmented by the addition of arc labels and transition inscriptions [6].

2.2.1 Asynchronous Product Automata

An APA consists of a family of so called *elementary automata* communicating by common components of their state (shared memory).

Definition 1 (Asynchronous Product Automaton (APA))**.**
An Asynchronous Product Automaton *consists of*

- *a family of* state sets $Z_s, s \in \mathbb{S}$,
- *a family of* elementary automata $(\Phi_t, \Delta_t), t \in \mathbb{T}$ *and*
- *a* neighbourhood relation $N : \mathbb{T} \to \mathfrak{P}(\mathbb{S})$

$\mathbb{S}$ and $\mathbb{T}$ are index sets with the names of state components and of elementary automata and $\mathfrak{P}(\mathbb{S})$ is the power set of $\mathbb{S}$. For each elementary automaton (Φ_t, Δ_t) with Alphabet *Φ_t, its* state transition relation *is $\Delta_t \subseteq \bigtimes_{s\in N(t)}(Z_s) \times \Phi_t \times \bigtimes_{s\in N(t)}(Z_s)$. For each element of Φ_t the state transition relation Δ_t defines state transitions that change only the state components in $N(t)$. An* APA's (global) *states* are elements of $\bigtimes_{s\in\mathbb{S}}(Z_s)$. To avoid pathological cases it is generally assumed that $N(t) \neq \emptyset$ for all $t \in \mathbb{T}$. Each APA has one *initial state* $q_0 = (q_{0s})_{s\in\mathbb{S}} \in \bigtimes_{s\in\mathbb{S}}(Z_s)$. In total, an APA $\mathbb{A}$ is defined by $\mathbb{A} = ((Z_s)_{s\in\mathbb{S}}, (\Phi_t, \Delta_t)_{t\in\mathbb{T}}, N, q_0)$. An elementary automaton (Φ_t, Δ_t) is *activated* in a state $p = (p_s)_{s\in\mathbb{S}} \in \bigtimes_{s\in\mathbb{S}}(Z_s)$ as to an *interpretation* $i \in \Phi_t$, if there are $(q_s)_{s\in N(t)} \in \bigtimes_{s\in N(t)}(Z_s)$ with $((p_s)_{s\in N(t)}, i, (q_s)_{s\in N(t)}) \in \Delta_t$. An activated elementary automaton (Φ_t, Δ_t) can execute a state transition and produce a *successor state* $q = (q_r)_{r\in\mathbb{S}} \in \bigtimes_{s\in\mathbb{S}}(Z_s)$, if $q_r = p_r$ for $r \in \mathbb{S} \setminus N(t)$ and $((p_s)_{s\in N(t)}, i, (q_s)_{s\in N(t)}) \in \Delta_t$. The corresponding state transition is $(p, (t, i), q)$.

APA provide the formal basis for the three syntactically different languages supported by the Simple Homomorphism Verification Tool (SHVT). Details are described in Section 2.3.

2.2.2 Petri Nets

Definition 2. *A net $N = (\mathbb{S}, \mathbb{T}, \mathbb{F})$ consists of:*

- *a finite set $\mathbb{S}$ of* places *(graphical symbol: ◯)*
- *a finite set $\mathbb{T}$ of* transitions *(graphical symbol: □)*

- *a* flow relation $\mathbb{F} \subset (\mathbb{S} \times \mathbb{T}) \cup (\mathbb{T} \times \mathbb{S})$.

The elements of $\mathbb{F}$ *are called arcs (graphical symbol:* $\rightarrow$*).*

The condition $\mathbb{F} \subset (\mathbb{S} \times \mathbb{T}) \cup (\mathbb{T} \times \mathbb{S})$ means that

- arcs lead from places to transitions (*input arcs*) or from transitions to places (*output arcs*); and
- at most, a single arc leads from a place (transition) to a transition (place).

The first definition of nets was given by C.A.Petri in his dissertation in 1962 [31].
Example:

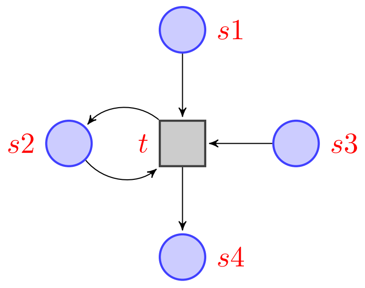


Figure 2.2: Example net

$\mathbb{S} = \{s1, s2, s3, s4\}, \mathbb{T} = \{t\}, \mathbb{F} = \{(s1,t), (s2,t), (s3,t), (t,s2), (t,s4)\}$
The places which are related by input arcs to a transition t are called *input places* of t. The set of these places is denoted with $^\bullet t$. Hence,
$^\bullet t := \{x \in \mathbb{S} | (x,t) \in \mathbb{F}\}$.
Similarly, the number $t^\bullet$ of all *output places* of t is defined as
$t^\bullet := \{y \in \mathbb{S} | (t,y) \in \mathbb{F}\}$.
In the considered example, the following holds
$^\bullet t = \{s1, s2, s3\}$ and $t^\bullet = \{s2, s4\}$.
In a net, states are depicted by a marking of the places and "dynamics" by change of that marking.

Definition 3. *A* marking of a net *is a mapping* $M : \mathbb{S} \rightarrow \mathbb{N}_0$*, where* $\mathbb{N}_0$ *is the set of all natural numbers, including the value 0. This means that each place may carry any given number of tokens.*

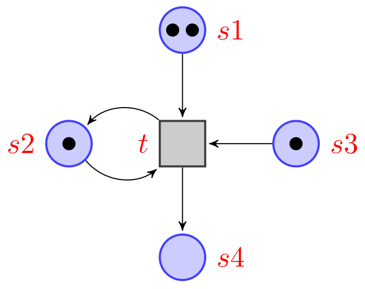


Figure 2.3: Example net with markings

In Fig. 2.3 the marking is given by: $M(s1) = 2, M(s2) = 1, M(s3) = 1, M(s4) = 0$
Changes in markings are effected by the occurrence of transitions.

Definition 4. *A transition t is* enabled *under a marking M, if each of its input places contains at least one token, i.e., if $M(x) \geq 1$ for all $x \in^{\bullet} t$.*

The transition t of the considered example is thereby thus enabled. An enabled transition can *occur*. If an transition enabled under marking M occurs, the marking M is replaced by a successor marking M'. As a consequence of that, the marking of the input places of the transition decrements by one token, the marking of the output places increments by one token and the marking of all other places of the net remains unchanged. Hence, the following holds:

Definition 5.

$$\begin{aligned} M'(x) &:= M(x) - 1 \textit{ for } x \in^{\bullet} t \setminus t^{\bullet}, \\ M'(x) &:= M(x) + 1 \textit{ for } x \in t^{\bullet} \setminus^{\bullet} t, \textit{ and} \\ M'(x) &:= M(x) \textit{ for all other } x \in \mathbb{S}. \end{aligned}$$

After the enabled transition of this example has occured, the marking of Figure 2.4 prevails.

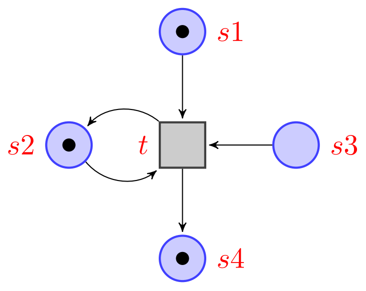


Figure 2.4: Example net after transition has occured

As apparent in Fig. 2.4 (place $s2$), a place may be input place as well as output place of a transition. The marking of such a place contributes towards the enabling condition of the corresponding transition, but the occurence of the transition does not change it. Under the marking resulting from the occurence of t in Fig. 2.4, the transition t is not enabled because place $s3$ contains no token (cf. Section 2.3.3).

2.3 Tool Support

In order to analyse the behaviour of a business or technical process with the PSA, an appropriate formal representation has to be chosen. In our approach, we choose an operational finite state model of the behaviour of the given process that is based on APA, a flexible operational specification concept for co-operating systems [30]. For the specification of the APA model we use PL and a compiler and interpreter which is provided by the SHVT.

The SHVT [30] is used to analyse APA models. It has been developed at the *Fraunhofer-Institute for Secure Information Technology*. The SHVT provides components for the complete cycle from formal specification to exhaustive validation as well as visualisation and inspection of computed reachability graphs and minimal automata. The applied specification method based on APA is supported by this tool. The tool manages the components of the model, allows to select alternative parts of the specification and automatically *glues* together the selected components to generate a combined model of the APA specification. After an initial state is selected, the Reachability Graph (RG) is automatically computed by the SHVT.

The tool furthermore provides an editor to define homomorphisms on action languages, it computes corresponding minimal automata [12] for the homomorphic images and checks simplicity of the homomorphisms.

The SHVT currently supports three specification methods:

1. State Transition Pattern notion of APA
2. PN
3. APA

All three methods use basic PL syntax which is introduced in Section 2.3.1.

Method (1) is the currently favoured specification method. This method is introduced in Section 2.3.2 and is later on used as a base for the EPC specifications introduced in Section 3.1 and applied for the Dam example in Section 5.1 and the Olympic Games example in Section 5.2.

Method (2) is introduced in Section 2.3.1 and later on used in Section 3.3 to convert the automatically discovered processes into operational models for the PSA.

Method (3) is introduced and used in Section 3.2 for the operational specification of processes in BPEL notation. This notation is not used in the examples from the MASSIF scenario but useful for the coverage of other possible process input.

Several papers describing the theoretical background are available at
`http://sit.sit.fraunhofer.de/smv/publications/`.

2.3.1 Introduction to PL

We will now use an example from the SHVT tutorial [29] to introduce the basic notion of the PL.

The objective of this section is to show the most important steps in specification and analysis by using this specification method.

Briefly said in this example data with a certain structure stored by agent A is transferred from agent A to agent B in a non deterministic sequence.

At first a project file for has to be created by starting the project manager from the PSA main menu. The used data structures and functions have to be specified in the so called preamble language of SHVT. To add a preamble file to a project the folder for the preamble has to be selected (mouse-l). In our example we can only select the displayed root node in the project tree. Afterwards the command File > Add Preamble(s) from the menu bar has to be executed. The new preamble file can be opened by executing the command Edit in the context menu of the newly created node in the project tree or by clicking mouse-m on the new node. The source code can be copied into the preamble editor using mouse-m if the text is selected :

```
defset tag = { 't1', 't2', 't3' };
defset data = { 'data' };
defset message = pro ( tag, data );
defset sequence = seq (message);
```

Compile the buffer with the command File > Compile Buffer. It's also possible to compile selected parts of this file using the command File > Compile Region. If no region is selected the definition at the current cursor position will be compiled. The functions can be tested in the preamble editor. For instance the following tests could be typed or copied into the test window below the editor pane:

```
('t1','data') ? Sequence;
sfind('t2','t1'.'t2');
sfind('t3','t1'.'t2');
```

The first call determines whether `('t1','data')` is an element of `Sequence`. The call of the predefined function `sfind` determines the position of the element `'t2'` and `'t3'` in the sequence `'t1'.'t2'`. `sfind` returns `0` if the element is not found. If the command Test > Test Buffer from menu bar is executed the following results are displayed in the message pane of the preamble editor:

```
('t1','data') ? sequence;  :=  (t1,data) element of sequence
sfind('t2','t1'.'t2'); := 2
sfind('t3','t1'.'t2'); := 0
```

It is possible to define own preamble functions. See [16] and for examples. If everything is OK the *Compile* switch for the preamble must be activated (mouse-l) to include this preamble into the code generation for analysis. The compile switch can also be used to toggle between different preamble variants. In [33] the development of a project for a MASSIF example is described in detail.

2.3.2 APA Transition Pattern

After defining the basic functions and sets the state transition pattern can be defined. State transition pattern are defined textual. So we add a new preamble file to our project. As the basic sets and functions will be used the file should be compiled after the preamble with the basic definitions. Therefore the node with the state transition pattern can be placed after the node with the basic sets and functions using drag and drop. After the order is OK create a link from the state transition pattern node to the basic preamble using the command Create Link in the context menu of the state transition pattern node and enter the state transition pattern code:

```
def_role A {Data : sequence := ('t1','data').('t2','data').('t3','data')};
def_role B {Data : sequence := :: };

def_state Network : sequence := ::;

def_trans_pattern A Send
    (t,d)
    (t,d) << A_Data,
    sfind(t,'t1'.'t2')>0,
    (t,d)>> Network;

def_trans_pattern B Receive
    (b)
```

```
  b << Network,
  b >> B_Data;
```

Two roles are defined for the two agents `A` and `B` (`def_role`). For every agent a local state component `Data` is defined, which stores the data to be transmitted respectively will be received. Data will be exchanged using the global state component `Network`.

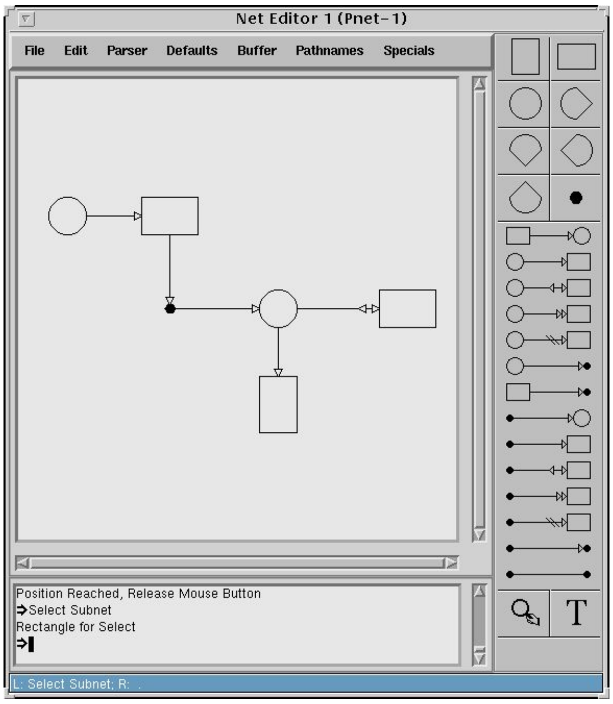
In state transition pattern notion some special operators for reading, writing and testing state components are available. Thus the transition pattern can be specified in a compact form. The transition pattern operators (like $>>$ and $<<$) are described in tabular form in [16]. Data is read and removed from state components using the $<<$ operator and added to the multiset of state components using the $>>$ operator. The function `sfind` is used to filter valid data.

2.3.3 Product Net (PN)

The PSA supports product nets (a special high-level Petri net class) for formal specification of process behaviour. Thus models specified in Petri Net Markup Language (PNML) [47] produced by process mining and discovery tools such as [44] can also be used instead of PL specifications (cf. Section 3.3).

PN result from augmenting the unlabelled nets by the addition of arc labels and transition inscriptions. For details please refer to [6].

For the input of PNs we have integrated the PN editor of the SHVT into the PSA. The graphical editor for PNs is described in [16].



```
Position Reached, Release Mouse Button
➔Select Subnet
Rectangle for Select
➔▌
```

L: Select Subnet; R: .

Figure 2.5: Graphical PN Editor

We plan to provide an PNML import interface in the tool adaptation work package if this is required by the scenario partners.

3 From Process Descriptions to Operational Models

3.1 Converting EPCs to Operational Model

To illustrate how PL can be used to model an EPC process with APA semantics we will first introduce the general idea and then give concrete examples from the MASSIF scenarios. Figures 3.1-3.4 show the basic objects for EPC modelling used in the presented examples. *or* connectors will not be considered in the following, since the semantics of this connector is not clear and there exist multiple interpretations [26]. In the case of the *or* connector the mapping of the EPC model has to be adapted to the concrete use case. For the other EPC connectors with clear semantics a standard mapping based on several implemented macros will be given. These pattern will enable the user to implement EPC models in PL event without expert knowledge of PL. Listing 3.1 contains the basic definitions for EPC implementation which will be explained below.

```
defset EPC_event_names = { e1, e2, e3, e4, e5, e6, e7, e8, e9 };

defset Connectors = { and, or, xor, nil  };

defset Event_seq = seq(EPC_event_names);

defset EPC_state_set = pro( Connectors : connector, Event_seq : events);
defconst  EPC_inactive   >> EPC_state_set = (nil, ::);

defcase update_state_fu : pro (EPC_state_set, EPC_state_set, EPC_event_names)
   >> EPC_state_set
   update_state_fu (state, init_value,event) =
        if connector(state) = nil then
           ( connector(init_value), event),
        if connector(state) = 'and' & sfind(event,(events(state))) = 0 then
           ( connector(state), imset(events(state).event))
        else_error  EPC_inactive();

def_role EPC
      {exit : bool := false }
      { active(state,input_events)
           ( connector(state) = 'xor' & l(events(state)) > 0) |
           ( connector(state) = 'and' & events(state) = events(input_events)) }
      { activate(state,init_value,event)
           state := update_state_fu(state,input_events, event) }
      { activate_if(output_state,init_value,event,test_event)
           when (event=test_event)  {
```

```
28               activate(output_state,init_value,event) } }
29       { deactivate(state)
30            state := EPC_inactive() }
31       { exit(event,test_event)
32             when (event = test_event) {
33                 exit := true } };;
```

Listing 3.1: Basic Macro definitions used for EPC implementation

In line 1 and 3 and a set of events used and the EPC connectors are defined. The set `Event_seq` defines a sequence of event names, to provide non-deterministic binding of events to variables of this type. The current state of an EPC is defined by the set `EPC_state_set` as a tuple of the corresponding input connector an the sequence of input events. The input connector value `nil` indicates that the corresponding EPC function is not active. The following macros defined for role EPC will be used in PL transition pattern implementations realising the EPC implementation for prediction of future states:

active (line 21) checks whether the EPC function with current state `state` and a possible sequence of input events assigned to `input_events` is active. When a EPC function becomes active, state must be initialised by the corresponding input connector and possible input events.

activate (line 24) tries to activate a EPC function based on the state of this function. The function `update_state_fu` implements the check, whether the function is activated.

activate_if (line 26) executes the macro `activate` if the selected event matches the connection to the function to be activated.

deactivate (line 29) deactivates a EPC function corresponding to `state`.

exit (line 31) must be executed for output events not connected to a function. If the "real" system state reaches this predicted state the EPC will be deactivated.

The application of these macros will be explained using the examples from Figures 3.1-3.4.

The naming conventions given in Table 3.1 will be used for every EPC function X in the PL code examples.

PL name	PL type	Usage
EPC X	Transition pattern	Computation of output event if transition pattern is active
EPC_X_input	Constant	Input events of X
EPC_X_output	Constant	Output events of X
EPC_X_state	State component	Current state of function X

Table 3.1: Naming conventions

Listing 3.2 shows the PL implementation of the EPC in Figure 3.1.

Thus for every EPC function a constant, which defines possible input events and the corresponding EPC connector, and a state component, which stores the current state of the EPC function have to be defined. See, e.g., Line 1 and 2 in Listing 3.2. The function `active` checks based on this values whether

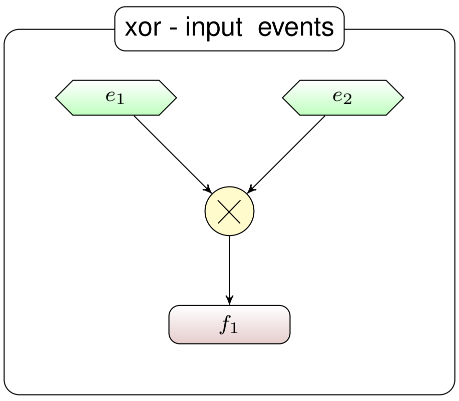


Figure 3.1: EPC *xor* - triggering events

the function is activated (Listing 3.1, line 21). The code for input events connected with function `f2` in Listing 3.3 differs to the *xor* example only in the value of the connector value of the input variable. These pattern do not express the topology of the EPC. The topology of the diagram will be specified by generation of output events in the transition pattern representing the EPC functions.

```
defconst  EPC_f1_input  >> EPC_state_set =  (xor, e1.e2);
def_state EPC_f1_active :  EPC_state_set := EPC_inactive();

def_trans_pattern EPC f1

   ...
   active(EPC_f1_active, EPC_f1_input()),
   ...
   deactivate(EPC_f1_active);
```

Listing 3.2: PL implementation of EPC Figure 3.1

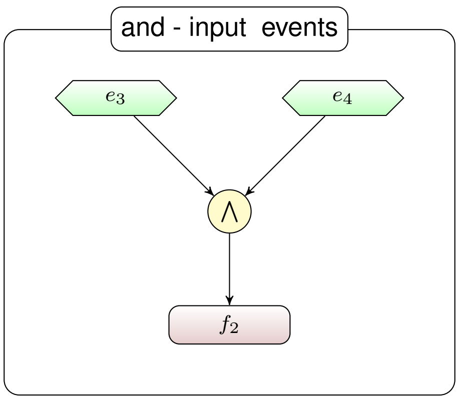


Figure 3.2: EPC *and* - triggering events

```
defconst  EPC_f2_input  >> EPC_state_set =  (and, e3.e4);
def_state EPC_f2_active :  EPC_state_set := EPC_inactive();

def_trans_pattern EPC f2
   ...
   active(EPC_f2_active, EPC_f2_input()),
   ...
   deactivate(EPC_f2_active);
```

Listing 3.3: PL implementation of EPC Figure 3.2

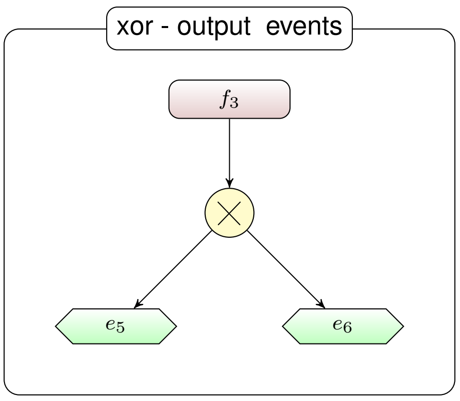


Figure 3.3: EPC *xor* - generating events

```
defconst  EPC_f3_output  >> EPC_state_set =  (xor, e5.e6);
def_state EPC_f3_active :  EPC_state_set := EPC_inactive();

def_trans_pattern EPC f3
    (con,event,events)
   ...
   (con,events) ? EPC_f3_output(),
   event ? events,
   activate_if(EPC_..._active,EPC_..._input(), event,'e4') },
   activate_if(EPC_..._active,EPC_..._input(), event,'e5') },
   deactivate(EPC_f3_active);
```

Listing 3.4: PL implementation of EPC Figure 3.3

```
def_state EPC_f4_active :  EPC_state_set := EPC_inactive();

def_trans_pattern EPC f4
    (con,event,events)
```

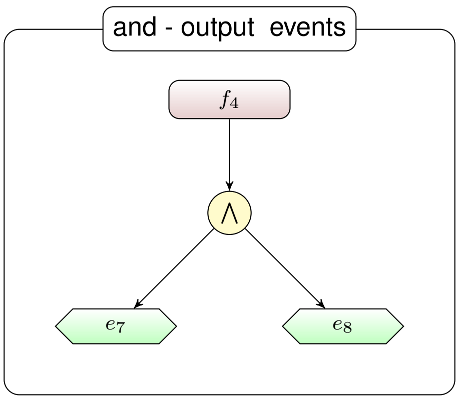


Figure 3.4: EPC *and* - generating events

```
    ...
    activate(EPC_..._active,EPC_..._input(), e7) },
    activate(EPC_..._active,EPC_..._input(), e8) };
    deactivate(EPC_f4_active);
```

Listing 3.5: PL implementation of EPC Figure 3.4

The parallel execution of functions as shown in Figure 3.5 can be realised similar to the schema presented for the four basic examples (see Listing 3.6). To achieve parallel synchronous execution the PL code for the two functions is included in one transition pattern. The output events will be generated for both functions as shown in the previous examples. Asynchronous execution could also be realised by implementing a auxiliary function generating the appropriate events.

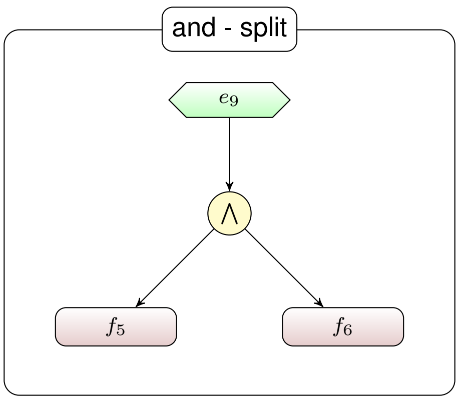


Figure 3.5: EPC parallel execution of functions

```
defconst  EPC_f5_input  >> EPC_state_set =  (and, e9);
defconst  EPC_f6_input  >> EPC_state_set =  (and, e9);
def_state EPC_f5_active :  EPC_state_set := EPC_inactive();
def_state EPC_f6_active :  EPC_state_set := EPC_inactive();
```

```

def_trans_pattern EPC f5_f6
   ...
   active(EPC_f5_active, EPC_f5_input()),
   active(EPC_f6_active, EPC_f8_input()),
   ...
   ...
   deactivate(EPC_f5_active),
   deactivate(EPC_f6_active);
```

Listing 3.6: PL implementation of EPC Figure 3.5

Beside the presented schema also a table oriented implementation supported by the PSA is possible. The advantage of this approach is the capability of dynamic EPC extension. The EPC can be changed directly by PL code or as described in Deliverable D4.1.3 [33]. The disadvantage is given by an more complicated implementation. Partitioning of the code between different transition pattern is not possible since there is only one pattern (cf. Listing 3.7). Thus extensions implemented in PL have to be part of this pattern. Therefore, static EPC models should be implemented using the transition pattern schema presented, while models with the necessity of dynamic EPC extensions can be realised by table specifications. The EPC table implementation will be explained on the example presented in Figure3.6 in Listing 3.7.

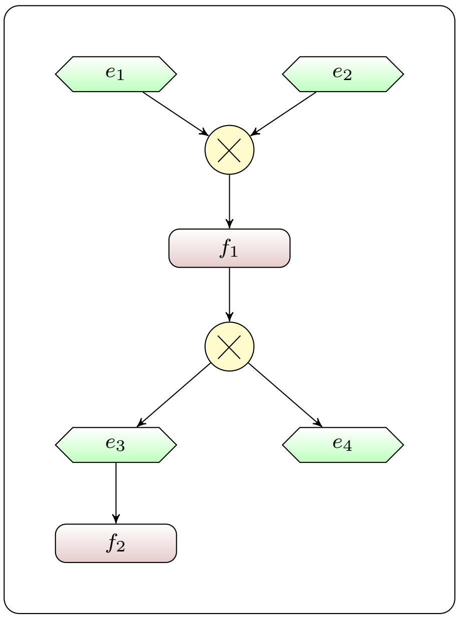


Figure 3.6: Example EPC to illustrate the table based specification

```
defset EPC_event_names = { e1, e2, e3, e4  };
defset EPC_functions = { f1, f2, exit };
defset EPC_connectors = { and, or, xor, nil  };
defset EPC_event_seq = seq(EPC_event_names);

defset EPC_state_set = pro( EPC_connectors : connector, EPC_event_seq : events);

defconst  EPC_inactive   >> EPC_state_set = (nil, ::);

defset Event_to_Function_set = pro (EPC_event_names : fu_event, EPC_functions : evt_function);

defset Event_Function_seq = seq (Event_to_Function_set);

defset EPC_output_set = pro( EPC_connectors : out_connector,
                             Event_Function_seq : events_functions);

defset EPC_transition = pro ( EPC_functions : epc_function,
                              EPC_state_set : epc_state,
                              EPC_state_set : epc_input,
                              EPC_output_set : epc_output );

defset EPC_transitions = [ EPC_transition ];

defconst EPC_init >> EPC_transitions =
  [
    (f1, (nil,::),
         (xor,e1.e2),
         (xor,(e3,f2).(e4,exit))) ];

def_state EPC_state : EPC_transitions :=  EPC_init();

defset EPC_state_seq = seq (EPC_transitions);

defset EPC_sim_transition = pro ( EPC_event_names : sim_event, EPC_transitions );
defset EPC_sim_transition_seq = seq (EPC_sim_transition);

def_trans_pattern EPC transition
       ()
       successor_states := epc_transition(EPC_state),
       (event,new_state) ? successor_states,
       EPC_state := new_state;
```

Listing 3.7: PL implementation of EPC Figure 3.6

3.2 APA Semantics for BPEL Processes' Specifications

To illustrate how PL 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 data types and state patterns we will use strings that are as close as possible to the corresponding tags and attribute values used in the WS-BPEL and WSDL definitions.

3.2.1 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 are *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**. BPEL depends on the use of WSDL. "For maximum interoperability and platform neutrality, WSDL prefers the use of XML Schema Definition (XSD) as the canonical type system, and treats it as the intrinsic type system" [1]. So the mechanisms described in Section 4.3 can be used to define mappings from BPEL data to PL data. Let us assume that the following PL types are created for variables of the type *xsd:string*:

$$pName1 = \{``t1\_Value1", ``t1\_Value2",\}$$
$$pName2 = \{``t2\_Value1", ``t2\_Value2",\}$$

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, used by the SHVT, we get the following specifications 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);
```

PL syntax for the `variable` data type in Section 3.2.1

[1] Source:http://www.w3.org/TR/wsdl

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 PL 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` characterises 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 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 each partner, supposed to take part in the whole process. If a 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

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

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

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

The definition of the `partnerLinkType` data type using the PL syntax is given in Section 3.2.1.

```
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;
```

PL syntax for the `partnerLinkType` data type in Section 3.2.1

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. We define one additional global state component for *event* messages. According to the PL 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 = ::;
```

PL syntax for the `partnerLinkType` data type in Section 3.2.1

3.2.2 State Transitions and Roles in the Operational Model

Roles and state transitions. To each WSDL `partnerLinkType`, defined of the workflow, and the workflow itself we assign a separate role. We specify one additional role `EventHandler` and each transition pattern, assigned to this role specifies each event that is handled by the event handler or a `onAlarm` WS-BPEL tag. Roles in PL are defined using the keyword **`def_role`**.

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

Role definition in PL syntax

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 automata) 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 or the occurrence of a fault or event (caught by an `eventHandler`). 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. If an error is returned by some of the partners and caught by `faultHandlers` then the corresponding state transition will occur.

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;
```

PL syntax used to define transition pattern

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 receive 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 global state components the process starts with. Since in the initial state of such process there are no messages sent or received by the partners we initialise the all global state components as the empty sequence $(::)$.

3.2.3 Example of an Operational Model of a BPEL Process

Figure 3.7 shows the structure of an APA modelling a simple centralised BPEL process. The circles represent state components and boxes are elementary automata. In the scenario 5 partners take part: *centralCoordinator* (the workflow engine), *commPartner1* (partner 1), *commPartner2* (partner 2), *commPartner3* (partner 3) and *commPartner4* (partner 4). Each partner taking part in the scenario is modelled by one elementary automaton that performs the partner's actions. The state components *centralCoordState1*, *centralCoordState2*, *centralCoordState3*, *centralCoordState4* store local data for the central coordinator of the process and the state components *localState1Partner1*, *localState2Partner1*, *localState3Partner1* store data available only to *partner1*. 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 *centralCoordinator* may change its *centralCoordState1* and *centralCoordState2* but cannot read or change the state component *localState1Partner1* of the automaton of *partner1*. The state components *commPartner1*, *commPartner2*, *commPartner3*, *commPartner4* are globally shared between all the partners participating in the process. They are used for communication (e.g. service calls) between each partner and the central partner. A message is sent by adding it to the content of one of the state components and received by removing it from the corresponding state. The state component *eventStream* is also globally shared and is used to transport triggered events to the services which have to receive them.

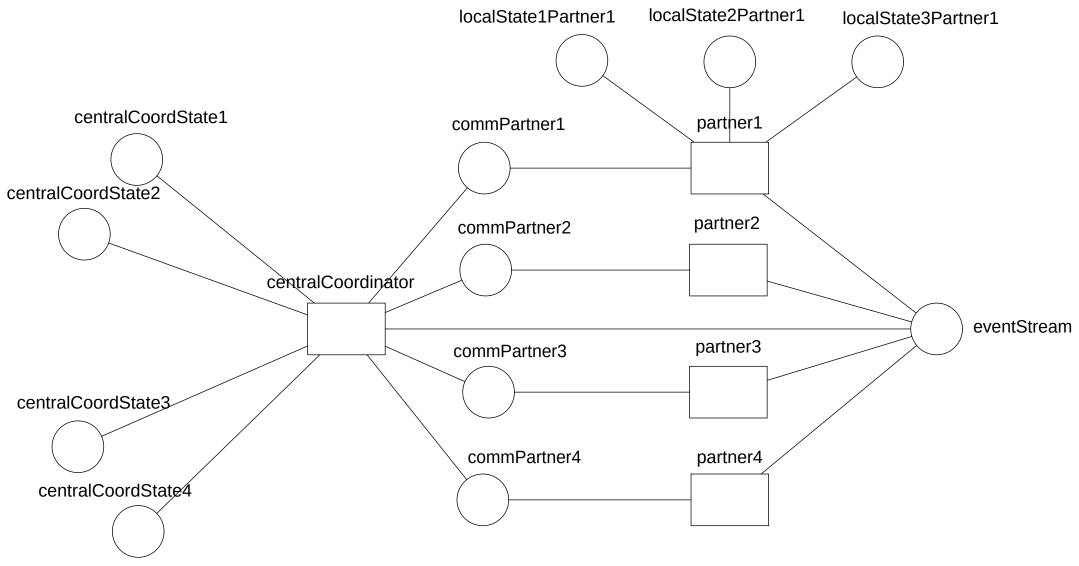


Figure 3.7: APA model of a BPEL process specification

3.3 Process Discovery

In order to assist the MASSIF scenario providers in specification of application processes, the PSA supports the use of process specifications, which have been generated by external tools. Specifically, it is possible to use a Petri net specification, which has been generated by the *ProM* [44] tool.

ProM [44] is a project by the process mining group at Eindhoven Technical University [1]. The current open source version of the ProM tool [2] uses Extensible Event Stream (XES) an XML-based standard for event logs as input. XES provides a format for the interchange of event log data between process mining tools and applications (cf. Figure 3.8 [2]). Tools like XESame [3] and Nitro [4] enable domain experts to specify how the event log should be extracted from existing systems and converted to XES [46].

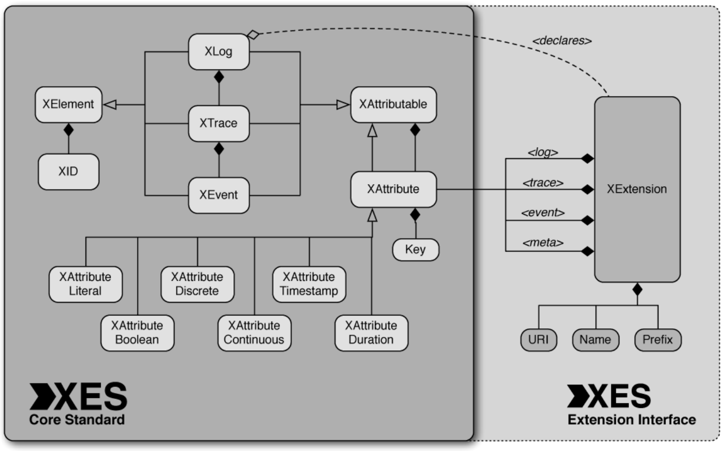


Figure 3.8: XES standard for event logs

As a proof of concept, we have used example data from the mobile money scenario and converted the logfile to XES format. Then we used the ProM tool with this logfile and discovered the process shown in Figure 3.9.

The transfer of this discovered process to the PSA was done manually using the graphical PN interface and resulted in the process description depicted in Figure 3.10.

[2]Source: http://www.xes-standard.org/
[3]http://www.processmining.org/xesame/start
[4]https://fluxicon.com/nitro/

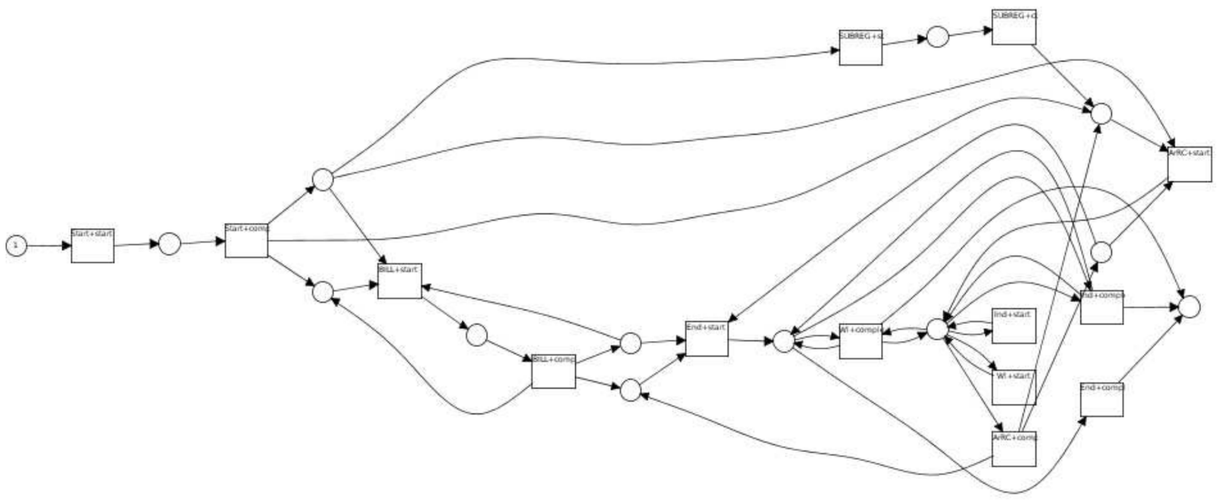


Figure 3.9: Process Discovery in FT Scenario

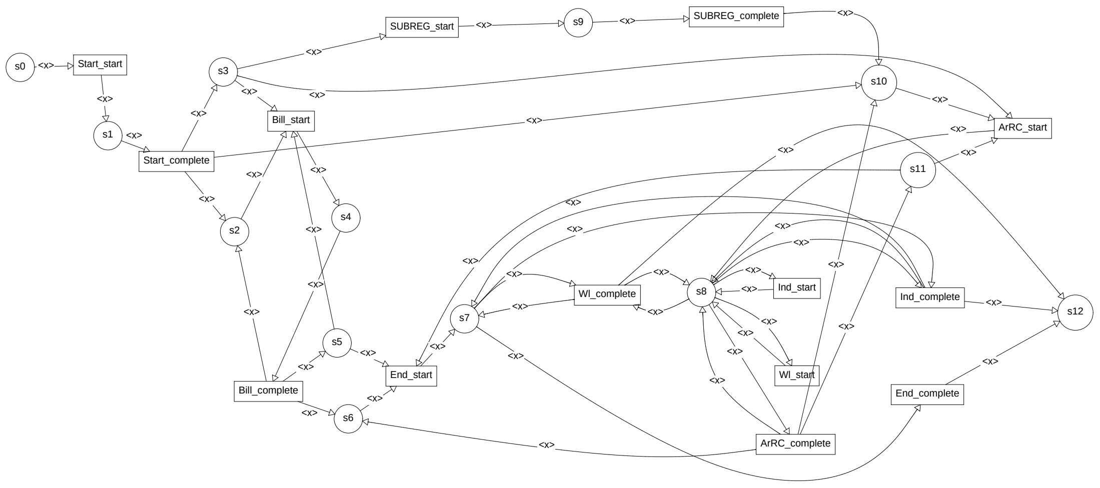
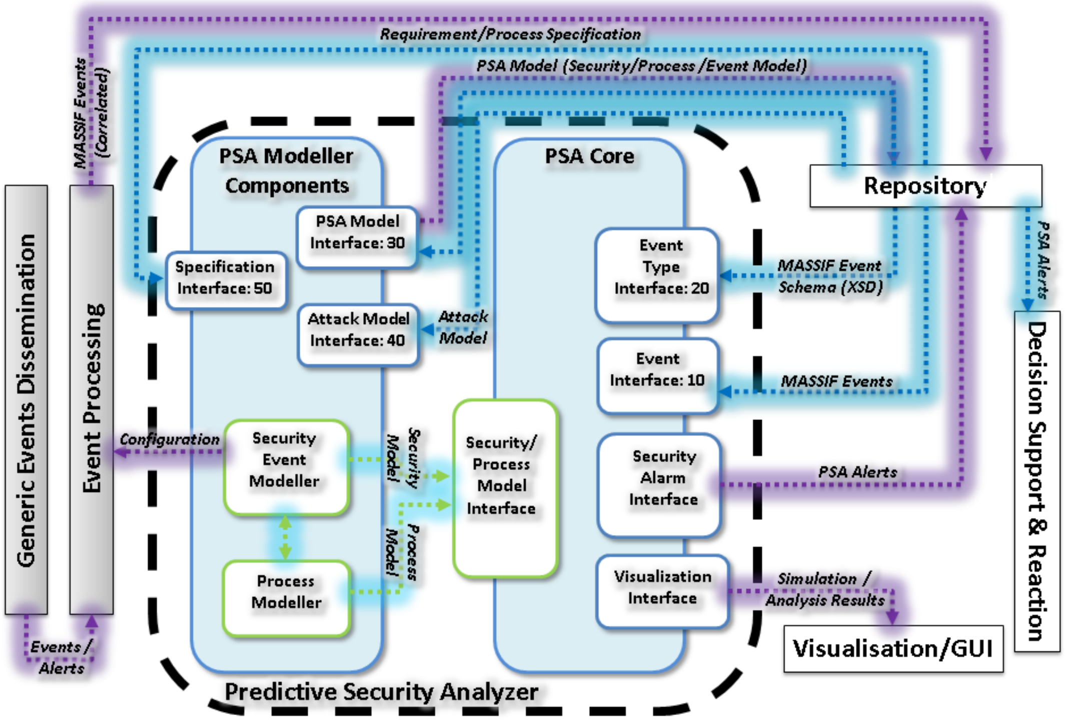


Figure 3.10: Discovered Petri Net from Fig. 3.9 transferred to PN

4 Simulation and Prediction

4.1 Architecture of PSA

The PSA block diagram is depicted in Figure 4.1. The quality of the performed analysis strongly depends on the quality and granularity of the process description as well as on the appropriate security event specifications.



Requirement/Process Specification
PSA Model (Security/Process/Event Model)
MASSIF Events (Correlated)
Generic Events Dissemination
Event Processing
Events / Alerts
Configuration
PSA Modeller Components
PSA Model Interface: 30
Specification Interface: 50
Attack Model Interface: 40
Attack Model
Security Event Modeller
Process Modeller
Security Model
Process Model
Security/ Process Model Interface
PSA Core
Event Type Interface: 20
Event Interface: 10
Security Alarm Interface
Visualization Interface
Predictive Security Analyzer
Repository
MASSIF Event Schema (XSD)
MASSIF Events
PSA Alerts
Simulation / Analysis Results
Visualisation/GUI
Decision Support & Reaction

Figure 4.1: Architecture of the Predictive Security Analyser

The PSA comprises two components to derive the specifications of security events and processes, namely the Security Event Modeller (SEM) and the PM components (cf. Figure 4.1).

Prior to the start of the engine, the process description and security goals/events will be transformed into PSA understandable models, which are going to be used for the continuous real-time analysis and close-future simulation. This will be done in the security event modeller and the process modeller components. They will communicate with the model repository, which contains the attack models of the Attack Modeling and Security Evaluation Component (AMSEC) tool [21] and the previous models composed by the modellers. The *security model* and *process model interfaces* will provide access to the repository for the PSA engine. The interpreted models will be imported into the PSA in the initialisation phase.

4.2 Reachability Analysis and Prediction of Behaviour

Reachability Graph of an APA Model. Formally, the behavior of our operational APA model of the business process is described by a Reachability Graph (RG). In the literature, this is sometimes also referred to as Labeled Transition System (LTS).

Definition 6 (reachability graph)**.** *The behavior of an* APA is represented by all possible coherent sequences of state transitions starting with initial state q_0. The sequence

$$(q_0, (t_1, i_1), q_1)(q_1, (t_2, i_2), q_2) \dots (q_{n-1}, (t_n, i_n), q_n)$$

with $i_k \in \Phi_{t_k}$, where Φ_{t_k} is the alphabet of the elementary automaton t_k, represents one possible sequence of actions of an APA. State transitions $(p, (t, i), q)$ may be interpreted as labeled edges of a directed graph whose nodes are the states of an APA: $(p, (t, i), q)$ is the edge leading from p to q and labelled by (t, i). The sub-graph reachable from q_0 is called *reachability graph (RG)* of an APA.

In code generated from the EPC process descriptions we used the set of possible output events of an action as the alphabet of the elementary automaton representing the action. So the interpretation i is the output event. An example (cf. Section 5.1 Figure 5.4) for a state transition is:

$$(M\text{-}2, (\mathit{Wait\_for\_change}, \mathit{event} = \mathit{close\_gate}), M\text{-}9).$$

The parameters of this state transition are state M-2, the tuple composed of the elementary automaton $\mathit{Wait\_for\_change}$ and its interpretation $\mathit{event} = \mathit{close\_gate}$, and the followup state M-9.

Reachability Graph of an Product Net (PN) Model. The complete dynamics of a marked net is depicted by means of the RG. A RG is a directed graph the nodes of which are markings of the net produced by the occurrence of transitions - starting from the initial marking. From a marking M an arc (labelled t) leads to M', if, and only if, the transition t is enabled under M and the occurrence of t generates a successor marking M'. Markings under which no transition is enabled are called dead markings. The derivation of RGs is called reachability analysis.

For the examples presented in this deliverable it was possible to compute the complete RG (cf. Section 5.1 and Section 5.2).

4.2.1 Prediction of close-future Process Actions

The process execution trace and the current state of the process is computed using the measured events.

As the process description and the data model of the events is formalised in the process model, a RG can be computed starting with the current state of the process. The RG contains possible future actions to be predicted. The abstraction level of the future events to be predicted determines the complexity of the computed RG. In Section 4.3 the mechanisms to map system events to appropriate PL code will be described. Section 4.2.2 will explain the possibilities to synchronise the simulation model with the state of the monitored process and the environment of this process, independent of the process events send to the PSA.

4.2.2 Synchronisation of Model State with Monitored Process

The PSA simulates processes in the MASSIF system on a more abstract level. This level has to be specified by an expert for the process domain. During runtime it's necessary to synchronise these two levels. Naturally, it can occur that the process and the abstraction level of the model do not fit together perfectly. Mechanisms to deal with the uncertainty of the combination of the concrete process and the simulation process level are explained in Section 4.4 and [33]. The handling of problems as unknown events, unexpected process behaviour, and missing events will be described there. For the start of an simulation the current system state (q_0, 4.2) has to be defined. From this state according to the adjusted simulation options the PSA computes a part of the RG as described in 4.2. Now events coming from the Event Interface (cf. Section 4.1) trigger the re-computation of this graph to keep the model state synchronous with the process and system state. The PSA provides receiving of two types of external events:

Events matching model events: These events will be mapped to the process model and the PSA tries to find successor states already computed to mark them as current process state. The methods to be applied in cases when no successor evnts are found are described in section 4.4.

Events changing the state: Changes of the environment which are relevant for the process model or the monitoring process, e.g., the patch level of a certain host could be part of the model and checked by the monitoring automaton or even tested in the process model. After the reception of such an event the state of the model will be changed according to the adjusted simulation options and the RG will be recomputed from the current system states.

4.3 Event Model

4.3.1 Feature Selection

The PSA is capable to select the relevant attributes of the events needed for simulation.

As an example, the selection (green check marks) of nodes in the XML tree shown in Figure 4.2 will reduce the XML-code in Listing 4.1 to the event shown in Listing 4.2. The produced output can be mapped to a PL list as shown in Listing 4.3. The PL standard function `getVal` provides access to the data described by the path of the selected attributes (cf. [16]). The mapping of the "real" data to PL code will be explained in the next section.

Besides the XML format described here, other formats (e.g., for the events from the MASSIF database) will be supported by the PSA prototype (Deliverable D4.2.3), which is due in Month 27 of the project.

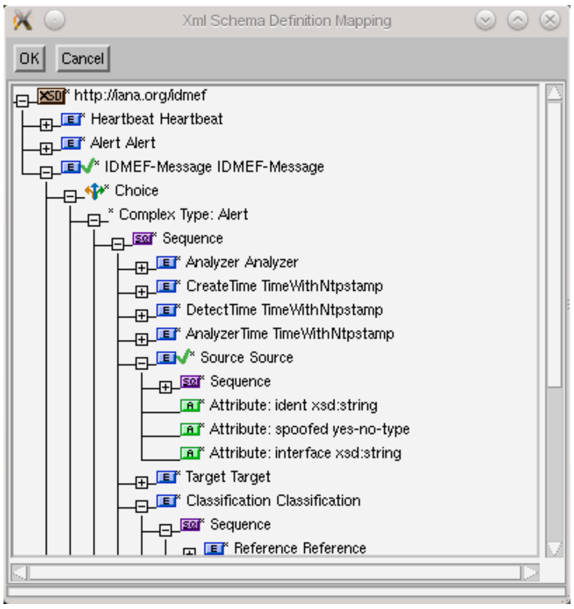

Xml Schema Definition Mapping
OK
Cancel
XSD * http://iana.org/idmef
E * Heartbeat Heartbeat
E * Alert Alert
E ✓ * IDMEF-Message IDMEF-Message
* Choice
* Complex Type: Alert
SQ * Sequence
E * Analyzer Analyzer
E * CreateTime TimeWithNtpstamp
E * DetectTime TimeWithNtpstamp
E * AnalyzerTime TimeWithNtpstamp
E ✓ * Source Source
SQ * Sequence
A * Attribute: ident xsd:string
A * Attribute: spoofed yes-no-type
A * Attribute: interface xsd:string
E * Target Target
E * Classification Classification
SQ * Sequence
E * Reference Reference

Figure 4.2: Feature selection

```
<idmef:IDMEF-Message version="1.0">
  <idmef:Alert messageid="abc123456789">
    <idmef:Analyzer analyzerid="bc-fs-sensor13">
        ....
    </idmef:Analyzer>
    <idmef:Source>
        XXXX
    </idmef:Source>
  </idmef:Alert>
</idmef:IDMEF-Message>
```

Listing 4.1: IDMEF input event

```
<idmef:IDMEF-Message version="1.0">
    <idmef:Source>
        XXXX
    </idmef:Source>
</idmef:IDMEF-Message>
```

Listing 4.2: Reduced IDMEF event

```
[IDMEF-Message [Source XXXX]]
```

Listing 4.3: PL data representing IDMEF event

4.3.2 Mapping, Normalisation and Transformation to PL

The PSA provides a Graphical User Interface (GUI) to define a mapping from events delivered to the PSA to internal PL notation of data. The PL language offers restricted data types: *numbers*, *symbols*, and *sequences*, *lists* and *tuples* of these basic data types (cf. [16]). To be able to handle information which is hard to express using the basic data types, the PSA has been extended to be able to convert arbitrary event data to PL symbols which could not be expressed in the original PL syntax. The only way to create such symbols is to use the corresponding mapping. The PSA creates the interface for the mapping for a XSD schema. The procedure how to integrate such a schema into a PSA project is described in a step by step tutorial in [33]. The XML data described by the schema will be mapped to PL lists representing the XML data in the mapping. To reduce the complexity of the RG computed in simulation only data really needed in the simulation or in generated alarms should be selected in the mapping. The default for all basic XML data will be to exclude the data from the mapping. Every data to be mapped to PL data has to be explicitly selected. The following default mappings are available:

Normalisation: A XML Nmtoken is any mixture of name characters [1]. PL syntax is more restrictive for the characters which can be used in identifiers: ’_’, ’$’, the set of digits { 0, 1, ..., 9 }, and the characters { a, b, ..., z, A, B, ... Z }. All characters of XML Nmtokens not in this list will be deleted by the normalisation function, except ’:’ and ’’. ’:’ and ’_’ which will be mapped to ’_’. The normalisation function will be applied to all XML tokens involved in the selected mapping. For XML enumerations normalisation will be the default mapping if the corresponding XML element is selected and the corresponding PL set will be generated.

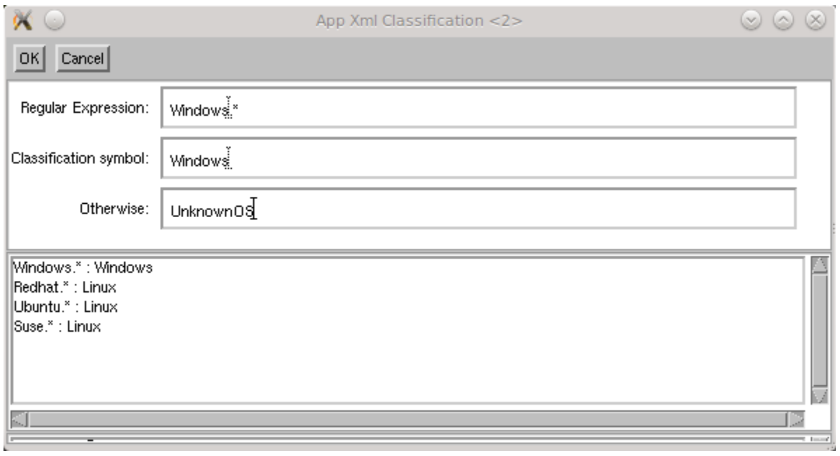
Use String: Any data will be converted to a PL symbol. The symbol will be presented in the original form without deletion of characters and can be bound to PL variables, but can’t be used as a PL constant in PL code. This mapping should be avoided if possible.

Classification: A list of regular expressions can be used to assign data to PL symbols. Figure 4.3 shows a simple example for such a mapping. Strings containing the distribution name or the OS name and additional information (“.*” in the regular expression) not relevant for simulation will be mapped to the OS name. Every pattern entered will be added to the list in the lower window. The regular expressions will be evaluated in the order of this list. A list item can be moved to the top of the list by selecting it with *mouse-l*. The corresponding PL set including the defined symbols will be generated. For XML elements of the type integer instead of regular expressions a range specification in the form $n1 - n2$ has to be entered.

Time to Universal Time: Converts the time format according to ISO (International Organization for Standardization) Representations of dates and times, 1988-06-15 (ISO 8601) to universal time [2].

The PSA provides the capability to load also other predefined mappings. The additional mappings can be put into the file patch.lisp in the installation directory or into the file .pnmrc in the home directory of the user who starts the PSA. Listing 4.4 shows some Lisp [41] code examples for such mappings. The first parameter of the function `add-mapping` defines the menu item for the PSA GUI. The second parameter defines a string which must match the XML type of the corresponding element. The Lisp implementation to the conversion of a XML data field to a PL symbol is defined in the third parameter.

[1] http://www.w3.org/TR/2000/REC-xml-20001006
[2] Non-negative integer - the number of seconds since midnight, January 1, 1900 Greenwich Mean Time (GMT) in seconds



Regular Expression: Windows.*
Classification symbol: Windows
Otherwise: UnknownOS

Windows.* : Windows
Redhat.* : Linux
Ubuntu.* : Linux
Suse.* : Linux

Figure 4.3: Mapping: classification menu

The implementation is defined in form of an anonymous function (lambda) with one parameter which is used to pass the XML data.

The fist mapping shows how to define a string classification without using the classification menu. The menu item “Check String Length” will be available in every context menu of XML strings. The second example (cf. Line 5), which can be used to extract seconds from Network Time Protocol (NTP) time stamps explains the usage of regular expressions in Lisp (cf. Function `reg-match`). The classification of date/time strings is explained by the third example in Line 14 [3].

```
1  (add-mapping
2   "Check String Length" "string"
3   #'(lambda (string) (string->pnet-symbol (if (> (length string) 100) "long_string "
   "normal_string"))))
4
5  (add-mapping
6   "Use NTP Seconds" "ntpstamp"
7   #'(lambda (ntp-timestamp)
8       (multiple-value-bind (ok match sec sec_fraction)
9           (reg-match "0x([A-Fa-f0-9]{4})([A-Fa-f0-9]{4}).0x[A-Fa-f0-9]{8}" ntp-timestamp)
10        (cond (ok
11               (read-from-string (format nil "#x~a" sec)))
12              (t (string->pnet-symbol "false"))))))
13
14 (add-mapping
15  "Office Time" "dateTime"
16  #'(lambda (date-time-string)
17      (let ((time (parse-iso8601 date-time-string)))
18        (cond (date-time-string
19               (let ((start  (get-universal-time (merge-date-times "08:00" time)))
20                     (end (get-universal-time (merge-date-times "18:00" time)))
21                     (utime (get-universal-time time))
22                     (wday (date-time-ywd-day time)))
23                 (cond ((and ((>= wday 1) (<= wday 5) (>= utime start) (<= utime end)))
25
```

[3]For more functions related to time objects see http://www.franz.com/support/documentation/current/doc/classes/date-time/date-time.htm

```
                          (string->pnet-symbol "true"))
                        (t (string->pnet-symbol "false")))))
              (t (string->pnet-symbol "error"))))))
```

Listing 4.4: Additional mappings

4.4 Behaviour in Uncertainty Situations

Normally, alarms produced by PSA will be generated by an monitor automaton as described in Section 4.5.1. Beside this, the PSA offers the option to generate alarms directly in the simulation using the PL standard function `alarm` which gets one PL symbol as a parameter. So unexpected situations which do not match the expected process behaviour can be modelled directly in the process model implemented in PL. In the current PSA version the alarm will only be logged on the console. The final format and the mechanisms for the delivery of alarms to the MASSIF system will be part of deliverable D4.2.3. The following cases related to uncertainty situations are handled by the PSA.

Missing events: Here, we consider the situation, when an event is detected, which is known, but not expected at this time in the current process state (as considered in the model).

If certain events which are included in the PSA model are not modelled, deadlocks in the simulation might occur. The PSA tries to solve this problem by trying to match later events with the computed RG. In this case, more than one current state might be computed. The depth of the search tree for such cases can be adjusted in the simulation options. Such cases can also occur, if the model does not fit to the process model. For such cases, semi-automatic model adaptation as described in [33] provides a solution. The decision what's the appropriate solution depends on whether the completeness of the event stream or the completeness of the model is more reliable. The best solution might be first to adjust the model ([33]) with a big training set and to use heuristics for recovery after this process.

Uncertain model state: During simulation it's possible that more than one current system state is determined. PSA tries to eliminate such states later in the prediction process if they do not match the incoming events.

Manual recovery: Missing events in combination with state elimination can lead to a divergence between process state and model state. The command *Change State* allows to adapt the model state. This solution is only applicable if it's possible to determine or to change (e.g. restart) the current process state.

Unknown Events: Here, we consider the situation, when an event is detected which is associated to the current process but out of scope of the reachability analysis. In this case a user decision is required. In case the process has changed, this should be reflected in the model by inserting an additional event in the process model. Otherwise an alert should be generated. In the first case, the modified process description will be used in further analysis.

This problem can be treated by the semi-automatic mechanisms described in [33].

For all problem cases of the list above, it's possible to produce alarms if the corresponding problem can't be solved. Also events, which produce the problem, can be ignored.

4.5 Monitoring Security Requirements

This section presents the *security analysis model* applied at runtime to identify security relevant states of the current state of the business process.

In addition to the predicted process behaviour, the security analysis model is needed in order to identify security critical situations. We use monitor automata as notation for our security analysis model to provide an operational specification of the security requirements model. These automata analyse the behaviour of the formalised process model during runtime and identify security relevant states of the formalised process model.

4.5.1 Monitor Automata

The monitor automaton concept has been developed to reflect attributes concerning all parameters of RG state transitions. Different security properties might be monitored simultaneously by allowing more than one transition of the monitor automaton to be triggered at the same time.

Formally, a *monitoring automaton* $\mathbb{M}$ consists of a set $\mathcal{M}$ of *labeled states*, an alphabet Λ of *predicates*, a *transition relation* $\mathcal{T}_{\mathcal{M}} \subseteq \mathcal{M} \times \Lambda \times \mathcal{M}$, and a set of initial states $\emptyset \neq \mathcal{M}_0 \subset \mathcal{M}$. Each state of the RG has an associated monitor automaton state set, which is computed during the simulation. $\mathcal{M}_0$ is assigned to the initial state q_0 of the RG under investigation. Each predicate $\lambda \in \Lambda$ of $\mathbb{M}$ is of the form $\lambda(x)$, where x is a state transition $(p,(t,i),q)$ of the RG. Each $\lambda \in \Lambda$ is associated with one of the transitions $\mathcal{T}_{\mathcal{M}}$ of $\mathbb{M}$.

During the run of the simulation, for each transition $(q_i,(t_j,i_k),q_l)$ of the RG, the monitor automaton state set for the RG state q_l is computed as follows:
Let $\mathcal{A}_i \subseteq \mathcal{M}$ be the monitor automaton's state set assigned to the current RG state q_i. The set $\mathcal{T}_{A_i}$ of transitions to be checked is now given by

$$\mathcal{T}_{A_i} = \{(m_m,\lambda_o,m_n) \in \mathcal{T}_M \mid m_m \in \mathcal{A}_i\}.$$

All predicates λ_o have to be checked for the current transitions of the RG $(q_i,(t_j,i_k),q_l)$. Based on the monitored transitions $\mathcal{MT}_l$ new monitor automaton states $\mathcal{B}_l$ and the changed monitor automaton states $\mathcal{C}_l$ are computed as follows:

$$\begin{aligned}\mathcal{MT}_l &:= \{(m_m,m_n) \in \mathcal{M} \times \mathcal{M} \mid (m_m,\lambda_o,m_n) \in \mathcal{T}_{A_i} \wedge \lambda_o((p_i,(t_j,i_k),q_l))\}\\ \mathcal{B}_l &:= \{m_n \in \mathcal{MT}_l | (m_m,m_n) \in \mathcal{MT}_l\}\\ \mathcal{C}_l &:= \{m_m \in \mathcal{MT}_l | (m_m,m_n) \in \mathcal{MT}_l\}\end{aligned}$$

For new states q_l, not already stored in the RG, the state set $\mathcal{A}_l$ is computed as follows:

$$\mathcal{A}_l := \mathcal{B}_l \cup (\mathcal{A}_i \setminus \mathcal{C}_l)$$

If the goal state q_l of the transition $(p_i,(t_j,i_k),q_l)$ has already been computed by another RG transition the new state set $\mathcal{A}_l$ has to be computed as follows:

$$\mathcal{A}_l := \mathcal{A}_i \cup \mathcal{B}_l$$

In this case the assignment of monitor automaton states to RG has to be updated starting from successor states of q_l and the associated new states N_l:

```
$N_l := (\mathcal{B}_l \setminus \mathcal{A}_i)$
$S := \{(m_{m_1}, q_l) \mid m_{m_1} \in N_l\}$
while $S \neq \emptyset$
    $S := S \setminus \{(m_{m_2}, q_{i_2})\}$
    $\mathcal{B}_{l_2} := \{m_{n_2} \in \mathcal{M} \mid (m_{m_2}, \lambda_o, m_{n_2}) \in \mathcal{T}_{A_{i_2}} \wedge \lambda_{o_2}((q_{i_2}, (t_{j_2}, i_{k_2}), q_{l_2}))\}$
    if $(\mathcal{B}_{l_2} \setminus \mathcal{A}_{i_2}) \neq \emptyset$
        $\mathcal{A}_{l_2} := \mathcal{A}_{i_2} \cup \mathcal{B}_{l_2}$
        $S := S \cup \{(m_{m_3}, q_{l_2}) \mid m_{m_3} \in (\mathcal{B}_{l_2} \setminus \mathcal{A}_{i_2})\}$
```

Thus, monitor automaton state sets are successively assigned to RG states during simulation. An example for an monitor automaton is given in Section 5.1 Figure 5.5.

5 Adapting Process Models for Simulation

In this chapter, we exemplify our approach with two different processes from the MASSIF scenarios.

5.1 Example: The Dam Scenario

The Dam example from Deliverable D4.2.1 [32] builds the base for the presented process models. Three example EPCs will be created:

- Monitoring of authorised staff in the control center (Figure 5.1).
- Monitoring of other staff present in the control center (Figure 5.2).
- Monitoring of gate processing (Figure 5.3).

This example shows the capabilities of the MASSIF framework. The monitoring and prediction part is divided between Generic Event Translation (GET) framework and PSA.

Things like counting staff members are handled by the GET framework (see [35]). Only the information what's the current role of staff present in the control station will be passed to the PSA. So EPCs related to the monitoring of the staff in the control center can be simplified to reduce complexity of the PSA simulation process.

The workflow of the gate control is specified in more detail. The decision of the chosen abstraction level and what details should be handled by the GET framework depends on the security requirements to be checked, on the information needs of produced alarms, on the complexity of the prediction process, and on the events to be simulated in the PSA.

During the specification process the developer has to find an appropriate balance between information needs for alarms and complexity of the model.

In our example, the EPC that models the gate control could be simplified with the disadvantage that the current state of the gate control process could not be part of the alarm. Just the information that there is a problem with the gate control could be delivered in this case. Details like the number of persons present in the control center would increase the size of the reachability graph produced by the PSA simulation in an unnecessary way.

Further reduction of complexity could be achieved by omitting the EPCs related to the control station staff. So there would be no prediction of the staff behaviour but it would be necessary to locate the monitoring into the GET framework and just to send the current state to PSA.

Listing 5.3 shows the PL implementation of the EPCs according to the presented translation schema.

This example can be compiled in the PSA and the complete reachability graph can be computed is this case. Since the computed graph is very small (19 nodes and 76 transition), prediction in this example can be reduced to finding the current state in this graph after reception of an external event by searching the EPC event name with the name stored in the PL interpretation variable `event`. All possible violations of security requirements defined in the corresponding monitor automaton will immediately be found. In more complex examples, only a part of the reachability graph will be computed and the graph will be extended according to the selected number of simulation steps. To avoid complete prediction of events related to the control center, the corresponding transitions pattern according to the EPC schema should not be part of the PL file. The GET framework could be configured to send an event, via the Complex Event Processing (CEP) engine and the SIEM database, with the current state of the control center to the PSA and by adjusting the PSA configuration this data could be stored into a PL state component. For instance GET could produce an event:

```
<control-station operator="active", other-staff="present" />
```

which could be stored automatically on a state component declared using `def_state` as a PL list:

```
[control_station [operator active] [other_staff present]]
```

Every change of this state component would trigger a re-computation of the reachability graph and the checking of the new transitions by the monitor automaton. So these events are excluded from prediction but the information stored in the state component can be used in predicates of the monitor automaton for the recomputed predicted EPC transitions.

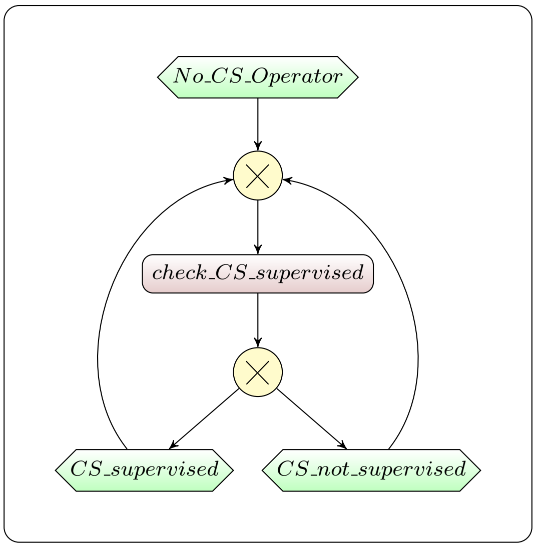


Figure 5.1: EPC control station operator

Listing 7.1 in the appendix 7.1 shows the PL code for the implementation of the three presented EPCs. The complete reachability graph can be computed for this model. The graph can be displayed using the PSA graph browser (see Figure 5.4). The names of the PL transition pattern causing the transitions are used by default as labels for the edges in the graph browser. The labels can be changed by using presentation homomorphisms (cf. [16]). Listing 5.1 shows the initial state `M-1` (named q_0 in 4.2) in textual form. The predecessor states and successor states (M-n) with the corresponding transition pattern are also part of this textual representation. The predecessor states are listed at the beginning, the successor states are listed after `M-1`, and the values of all state components are listed at the end.

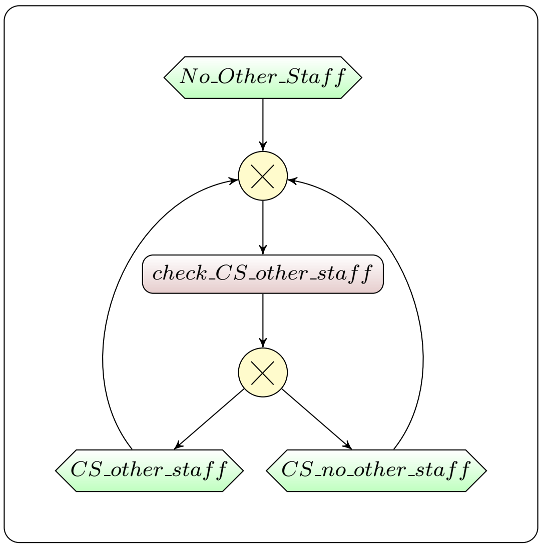


Figure 5.2: EPC control station other staff

```
M-9 EPC_wait_for_closed M-4 EPC_check_CS_operator_out M-3 EPC_check_CS_other_staff_out
M-1
EPC_check_CS_operator_in M-4  EPC_check_CS_other_staff_in M-3
EPC_wait_for_open M-2

EPC_check_CS_operator_in_active: <(and,CS_no_operator)>
EPC_check_CS_operator_out_active: <(nil,::)>
EPC_check_CS_other_staff_in_active: <(and,CS_no_other_staff)>
EPC_check_CS_other_staff_out_active: <(nil,::)>
EPC_exit: <false>
EPC_wait_for_change_active: <(nil,::)>
EPC_wait_for_close_active: <(nil,::)>
EPC_wait_for_closed_active: <(nil,::)>
EPC_wait_for_command_active: <(nil,::)>
EPC_wait_for_open_active: <(and,Gate_closed)>
```

Listing 5.1: Initial state of Dam reachability-graph

The security requirements that certain actions of the Dam workflows have to be supervised will be controlled by an monitor automaton as introduced in 4.5.1. Figure 5.5 shows a PSA specification of such an automaton. The initial state of the automaton `CS_not_supervised` is marked by the filled circle. The critical state `Action_not_supervised` is marked by the circle with the smaller filled circle inside. If this state is reached during prediction an alarm will be produced. The predicates attached to the arcs of the automaton define predicates for transitions of the reachability graph according to the syntax for edge search predicates defined ins [16]. This automaton is scheduled according to the algorithm presented in 4.5.1 during the computation of the reachability graph in the prediction process. The λ predicates in this section correspond to the predicates of the arcs in this automaton.

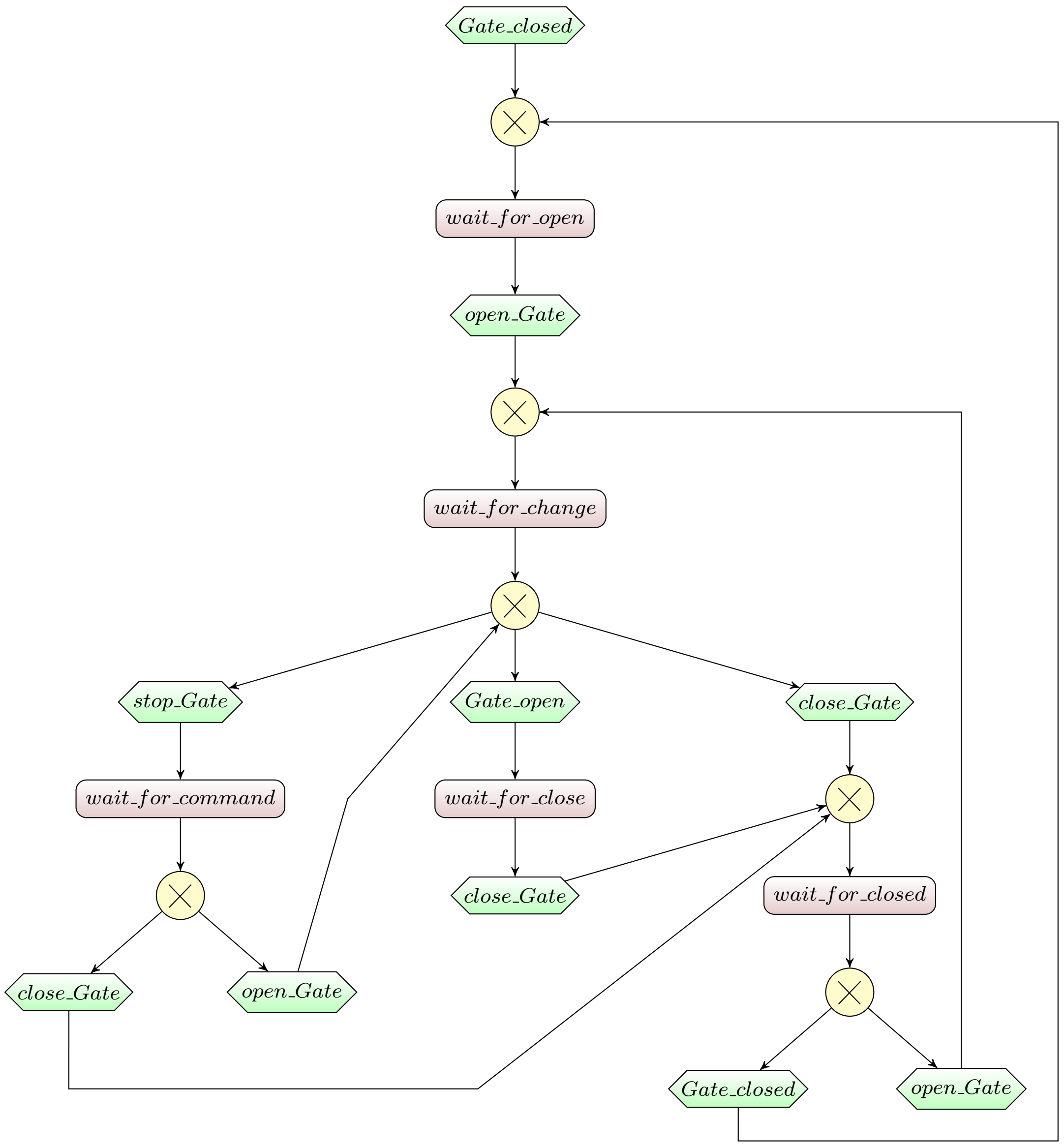


Figure 5.3: EPC gate control

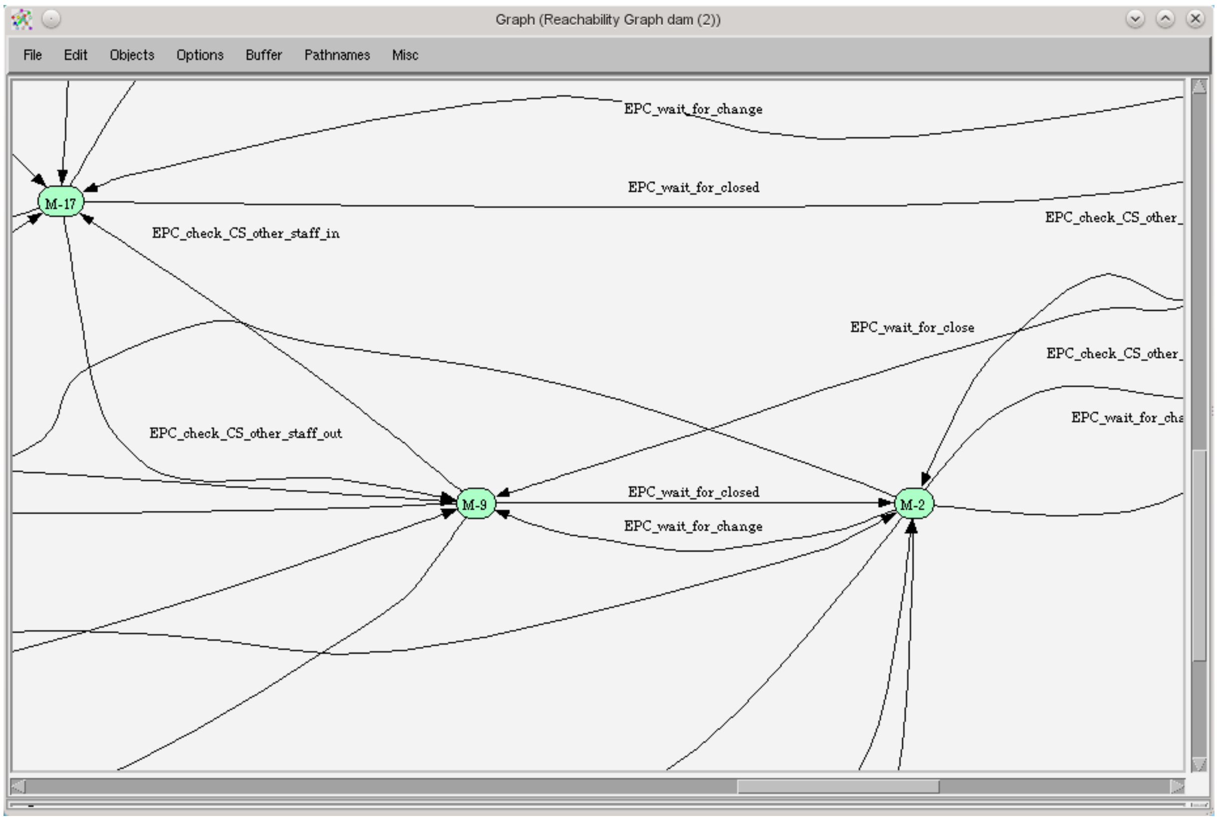


Figure 5.4: Reachability graph Dam

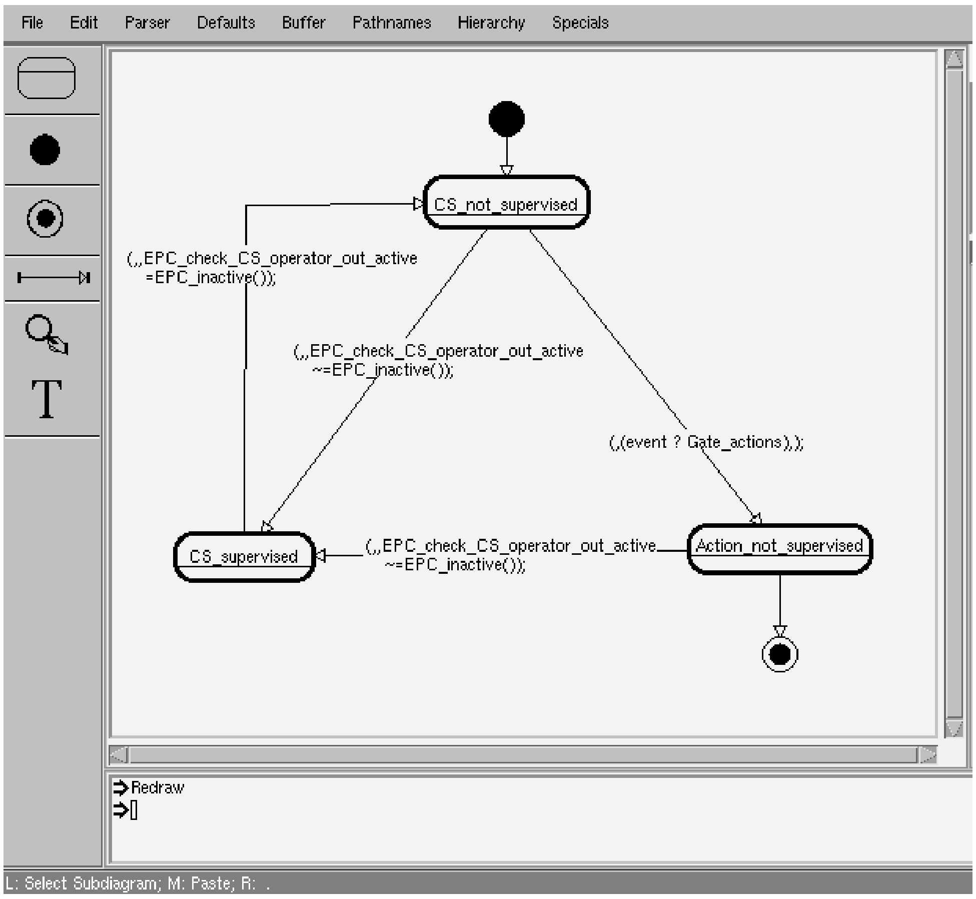

File
Edit
Parser
Defaults
Buffer
Pathnames
Hierarchy
Specials
CS_not_supervised
(,,EPC_check_CS_operator_out_active
=EPC_inactive());
(,,EPC_check_CS_operator_out_active
~=EPC_inactive());
(,(event ? Gate_actions),);
CS_supervised
(,,EPC_check_CS_operator_out_active
~=EPC_inactive());
Action_not_supervised
Redraw
L: Select Subdiagram; M: Paste; R: .

Figure 5.5: Monitor automaton Dam

5.2 Example: An Olympic Games Scenario

The EPC provided by the scenario provider does not correspond directly to a “business process” but shows an alternative application of the PSA. The EPC is based on “syslog” events in combination with the monitoring of control actions of the system behaviour. The PSA can be used to detect deviations from the specified behaviour and to predict the a possible “escalation” of the control process. The PL EPC implementation is given in Listing 7.2 in the appendix 7.2 according to the presented schema for the EPC implementation. It’s possible to compute the complete reachability graph for this model. The graph can be displayed restricted to the binding of the variable `event` with the graph browser of the PSA (cf. Figure 5.6). This graph directly corresponds to the EPC model since no constructs leading to branches in the RG are used. An alternative implementation for better usage of PSA features relating to uncertainty is given in [33].

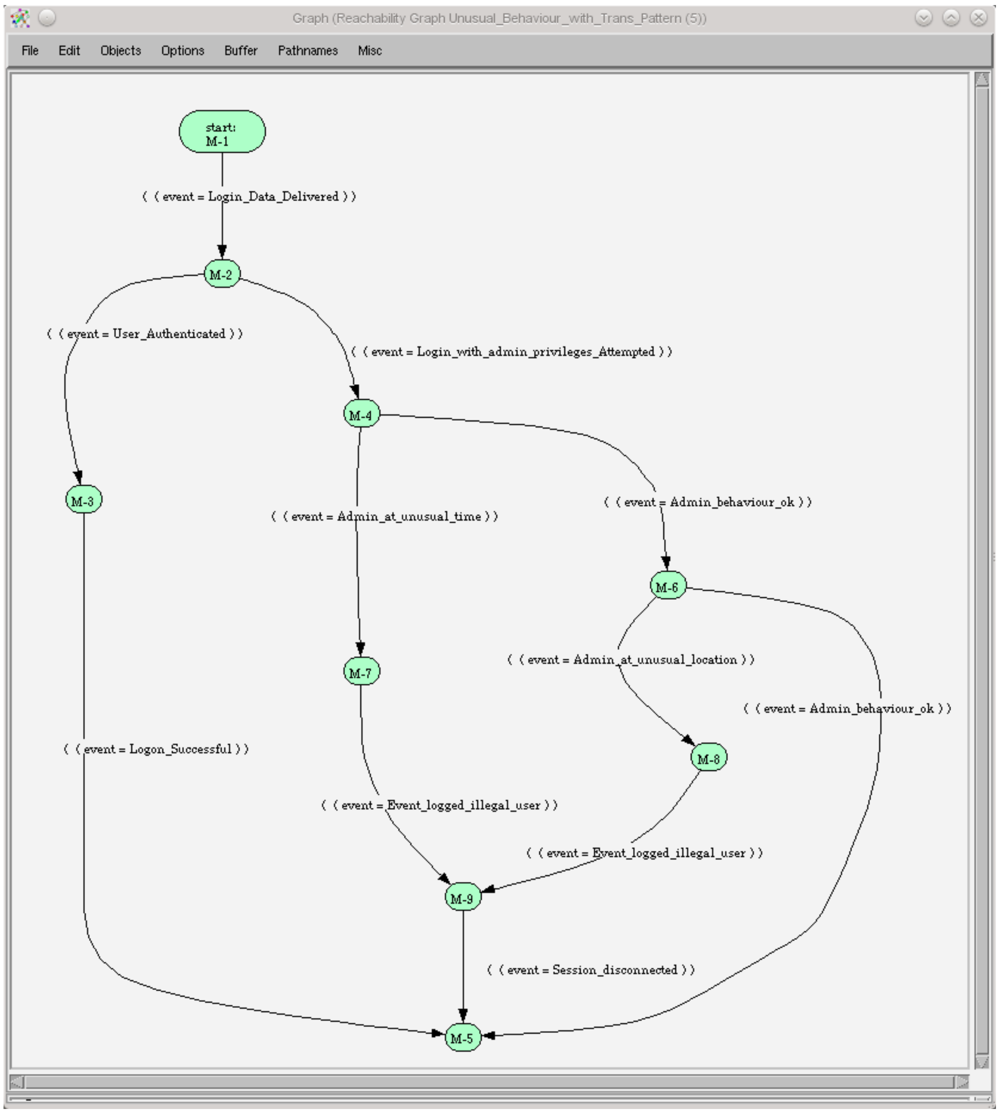

Graph (Reachability Graph Unusual_Behaviour_with_Trans_Pattern (5))
File
Edit
Objects
Options
Buffer
Pathnames
Misc
start: M-1
((event = Login_Data_Delivered))
M-2
((event = User_Authenticated))
((event = Login_with_admin_privileges_Attempted))
M-4
M-3
((event = Admin_at_unusual_time))
((event = Admin_behaviour_ok))
M-6
M-7
((event = Admin_at_unusual_location))
((event = Admin_behaviour_ok))
((event = Logon_Successful))
M-8
((event = Event_logged_illegal_user))
((event = Event_logged_illegal_user))
M-9
((event = Session_disconnected))
M-5

Figure 5.6: Reachability Olympic Games

6 Conclusion

This deliverable is part of the work package WP4.2. The particular objectives of this work package are, (1) to specify an executable event-driven process model triggered by real-time events, (2) to develop methodologies for performing dynamic predictive process analysis at runtime, (3) to provide techniques, featuring the ability to perform intensive simulation analysis under given hypothesis, and (4) to implement the provided techniques in an intelligent security event-processing engine.

This deliverable provides the results of the Tasks T4.2.2 "Process Model" and T4.2.3 "Dynamic Simulation and Analysis Methods". This comprises a description of a formal security-aware process model; specification of techniques for the identification of close-future security-threatening process states and simulation analysis methods, which inspect the behaviour of complex/parallel processes under given hypotheses. Therefore, the formalised process model consumes events from the runtime environment and reflects the current process state. A security analysis model, which is formalised by monitor automata, identifies current and close-future violations of the security requirements. A mapping model maps transitions in the security analysis model onto security alerts that are fed via the MASSIF repository service into the runtime environment. The alerts generated from these models at runtime can be used by other MASSIF components in order to enable anticipatory impact analysis, decision support and impact mitigation by adaptive configuration of countermeasures. Parts of the concept have been presented at the models@runtime workshop in [11].

Roadmap for future work. The last task of WP4.2 is the task T4.2.4 "Predictive Security Analyser". The analyser provided from this task will implement the techniques and methods provided in this deliverable. It will be able to perform a dynamic runtime analysis and an intensive simulation analysis of the project scenarios. Thus it will assist and support the SIEM framework, developed in the MASSIF project, in making important decisions concerning countermeasures and reactions against upcoming security threats.

An early prototype of the PSA is currently being tested and an interfaces to The Intrusion Detection Message Exchange Format (IDMEF) has been developed. Parts of the simulation and analysis methods are already present in the prototype, which has been delivered in D4.1.3. This prototype has been tested without the MASSIF environment using a mockup implementation of the adapter of the event interface. An example process description from Olympic Games scenario has been used to create test data.

The final PSA implementation is subject of deliverable D4.2.3 in M27. Visualisation of the current simulation state together with a graphical EPC representation will be developed and implemented. Import of PNML models as created by process discovery tools will be implemented.

Bibliography

[1] Process Mining Group, Eindhoven University of Technology Website, 2012.

[2] ProM process mining workbench Website, 2012.

[3] Alessandro Armando, Enrico Giunchiglia, Marco Maratea, and Serena Elisa Ponta. An action-based approach to the formal specification and automatic analysis of business processes under authorization constraints. *Journal of Computer and System Sciences*, 78(1):119–141, 2012.

[4] W. Arsac, L. Compagna, G. Pellegrino, and S. Ponta. Security Validation of Business Processes via Model-Checking. In *Engineering Secure Software and Systems (ESSoS 2011)*, volume 6542 of *LNCS*, pages 29–42. Springer, 2011.

[5] Lujo Bauer, Jarred Ligatti, and David Walker. More enforceable security policies. In *Foundations of Computer Security (FCS 2002)*, 2002.

[6] H. J. Burkhardt, P. Ochsenschläger, and R. Prinoth. Product Nets – A Formal Description Technique for Cooperating Systems. GMD-Studien 165, Gesellschaft für Mathematik und Datenverarbeitung (GMD), Darmstadt, September 1989.

[7] MASSIF Consortium. *Deliverable D2.1.1 - Scenario requirements*. Project MASSIF EC FP7-257475, April 2011.

[8] N. Delgado, A.Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering*, 30(12):859–872, 2004.

[9] Remco M. Dijkman. Diagnosing differences between business process models. In *Business Process Management (BPM 2008)*, volume 5240 of *LNCS*, pages 261–277. Springer, 2008.

[10] Remco M. Dijkman, Marlon Dumas, and Chun Ouyang. Semantics and analysis of business process models in BPMN. *Information and Software Technology*, 50(12):1281–1294, 2008.

[11] Jörn Eichler and Roland Rieke. Model-based Situational Security Analysis. In *Workshop on Models@run.time*, volume 794, pages 25–36. CEUR, 2011.

[12] S. Eilenberg. *Automata, Languages and Machines*, volume A. Academic Press, New York, 1974.

[13] A. Evesti, E. Ovaska, and R. Savola. From security modelling to run-time security monitoring. In *European Workshop on Security in Model Driven Architecture (SECMDA 2009)*, pages 33–41. CTIT, 2009.

[14] R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *Future of Software Engineering*, pages 37–54. IEEE, 2007.

[15] Ganna Frankova, Magali Seguran, Florian Gilcher, Slim Trabelsi, Jörg Dörflinger, and Marco Aiello. Deriving business processes with service level agreements from early requirements. *Journal of Systems and Software*, 84(8):1351–1363, 2011.

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

[17] Andreas Fuchs and Roland Rieke. Identification of Security Requirements in Systems of Systems by Functional Security Analysis. In *Architecting Dependable Systems VII*, volume 6420 of *LNCS*, pages 74–96. Springer, 2010.

[18] Raman Kazhamiakin, Marco Pistore, and Luca Santuari. Analysis of communication models in web service compositions. In *World Wide Web (WWW 2006)*, pages 267–276. ACM, 2006.

[19] G. Keller, M. Nüttgens, and A.-W. Scheer. Semantische Prozeš̈modellierung auf der Grundlage ”Ereignisgesteuerter Prozessketten (EPK)”. *Veröffentlichungen des Instituts für Wirtschaftsinformatik (IWi), Universität des Saarlandes*, 89, 1992.

[20] Richard Kissel. Glossary of key information security terms. NIST Interagency Reports NIST IR 7298 Revision 1, National Institute of Standards and Technology, February 2011.

[21] Igor Kotenko, Andrey Chechulin, and Evgenia Novikova. Attack modelling and security evaluation for security information and event management. In *SECRYPT*, pages 391–394, 2012.

[22] Fabrizio Maria Maggi, Marco Montali, Michael Westergaard, and Wil M. P. van der Aalst. Monitoring business constraints with linear temporal logic: An approach based on colored automata. In Stefanie Rinderle-Ma, Farouk Toumani, and Karsten Wolf, editors, *BPM*, volume 6896 of *Lecture Notes in Computer Science*, pages 132–147. Springer, 2011.

[23] Thierry Massart and Cédric Meuter. Efficient online monitoring of LTL properties for asynchronous distributed systems. Technical report, Université Libre de Bruxelles, 2006.

[24] David W. McCoy. Business Activity Monitoring: Calm Before the Storm. Gartner Research, 2002.

[25] M. Melik-Merkumians, T. Moser, A. Schatten, A. Zoitl, and S. Biffl. Knowledge-based runtime failure detection for industrial automation systems. In *Workshop Models@run.time*, pages 108–119. CEUR, 2010.

[26] Jan Mendling. *Metrics for Process Models: Empirical Foundations of Verification, Error Prediction, and Guidelines for Correctness*, volume 6 of *LNBIP*. Springer, 2008.

[27] Brice Morin, Tejeddine Mouelhi, Franck Fleurey, Yves Le Traon, Olivier Barais, and Jean-Marc Jézéquel. Security-driven model-based dynamic adaptation. In *Automated Software Engineering (ASE 2010)*, pages 205–214. ACM, 2010.

[28] M. Netjes, H.A. Reijers, and W.M. P van der Aalst. Supporting the BPM life-cycle with FileNet. In *Exploring Modeling Methods for Systems Analysis and Design (EMMSAD 2006)*, pages 497–508. Namur University Press, 2006.

[29] P. Ochsenschläger, J. Repp, and R. Rieke. *Simple Homomorphism Verification Tool – Tutorial*. Fraunhofer Institute for Secure Telecooperation SIT, Darmstadt, 2002.

[30] Peter Ochsenschläger, Jürgen Repp, Roland Rieke, and Ulrich Nitsche. The SH-Verification Tool Abstraction-Based Verification of Co-operating Systems. *Formal Aspects of Computing*, 10(4):381–404, 1998.

[31] C. A. Petri. Kommunikation mit Automaten. Dissertation, TH Darmstadt, 1962.

[32] Jürgen Repp and Roland Rieke. *D4.2.1 – Formal Specification of Security Properties*. FP7-257475 MASSIF European project, September 2011.

[33] Jürgen Repp, Maria Zhdanova, and Roland Rieke. *D4.1.3 – Methods and tools for reasoning about security with uncertain knowledge*. FP7-257475 MASSIF European project, September 2012.

[34] Roland Rieke and Zaharina Stoynova. Predictive security analysis for event-driven processes. In *Computer Network Security*, volume 6258 of *LNCS*, pages 321–328. Springer, 2010.

[35] Luigi Romano, Salvatore D'Antonio, Ivano Alessandro Elia, and Valerio Formicola. *D3.4.2 – Design of cross-layer event filtering, aggregation, correlation and abstraction*. FP7-257475 MASSIF European project, March 2012.

[36] A. Rozinat, M. T. Wynn, W. M. P. van der Aalst, A. H. M. ter Hofstede, and C. J. Fidge. Workflow simulation for operational decision support. *Data & Knowledge Engineering*, 68(9):834–850, 2009.

[37] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

[38] M. Seguran, C. Hebert, and G. Frankova. Secure workflow development from early requirements analysis. In *European Conference on Web Services (ECOWS 2008)*, pages 125–134. IEEE, 2008.

[39] C. Serban and B. McMillin. Run-time security evaluation (RTSE) for distributed applications. In *Symposion on Security and Privacy*, pages 222–232. IEEE, IEEE, 1996.

[40] George Spanoudakis, Christos Kloukinas, and Kelly Androutsopoulos. Towards security monitoring patterns. In *Symposium on Applied computing (SAC 2007)*, pages 1518–1525. ACM, 2007.

[41] Guy L. Steele. *Common LISP: The Language*. Digital Press, Bedford, MA, 2. edition, 1990.

[42] S. Tjoa, S. Jakoubi, G. Goluch, G. Kitzler, S. Goluch, and G. Quirchmayr. A formal approach enabling risk-aware business process modeling and simulation. *IEEE Transactions on Services Computing*, 4(2):153–166, 2011.

[43] T. Tsigritis and G. Spanoudakis. Diagnosing runtime violations of security & dependability properties. In *Software Engineering and Knowledge Engineering (SEKE 2008)*, pages 661–666. KSI, 2008.

[44] W. M. P. van der Aalst, B. F. van Dongen, C. Günther, A. Rozinat, H. M. W. Verbeek, and A. J. M. M. Weijters. Prom: The process mining toolkit. In *BPM 2009 Demonstration Track*, volume 489, pages 1–4. CEUR, 2009.

[45] Wil M. P. van der Aalst. Formalization and verification of event-driven process chains. *Information & Software Technology*, 41(10):639–650, 1999.

[46] H.M.W. Verbeek, JoosC.A.M. Buijs, BoudewijnF. Dongen, and WilM.P. Aalst. Xes, xesame, and prom 6. In Pnina Soffer and Erik Proper, editors, *Information Systems Evolution*, volume 72 of *Lecture Notes in Business Information Processing*, pages 60–75. Springer Berlin Heidelberg, 2011.

[47] Michael Weber and Ekkart Kindler. The petri net markup language. In *Petri Net Technology for Communication-Based Systems*, volume 2472 of *LNCS*, pages 124–144. Springer, 2003.

[48] Komminist Weldemariam and Adolfo Villafiorita. Procedural security analysis: A methodological approach. *Journal of Systems and Software*, 84(7):1114–1129, 2011.

[49] C. Wolter, M. Menzel, A. Schaad, P. Miseldine, and C. Meinel. Model-driven business process security requirement specification. *Journal of Systems Architecture*, 55(4):211–223, 2009.

7 Appendix A

7.1 Implementation of Dam Example

```
defset Event_names = { Gate_closed, open_Gate, stop_Gate, close_Gate, Gate_open, Gate_closed,
                       CS_no_operator, CS_operator,
                       CS_no_other_staff, CS_other_staff };

defset Connectors = { and, or, xor, nil  };

defset Event_seq = seq(Event_names);

defset EPC_state_set = pro( Connectors : connector, Event_seq : events);
defconst  EPC_inactive   >> EPC_state_set = (nil, ::);

defcase update_state_fu :pro (EPC_state_set, EPC_state_set, Event_names) >> EPC_state_set
   update_state_fu (state, init_value,event) =
        if connector(state) = nil then
           ( connector(init_value), event),
        if connector(state) = 'and' & sfind(event,(events(state))) = 0 then
           ( connector(state), imset(events(state).event))
        else_error  EPC_inactive();

def_role EPC
      {exit : bool := false }
      { active(state,init_value)
           (connector(state) = 'xor' & l(events(state)) > 0) |
           ( connector(state) = 'and' & events(state) = events(init_value)) }
      { activate(state,init_value,event)
          state := update_state_fu(state,init_value, event) }
      { activate_if(output_state,init_value,event,test_event)
          when (event=test_event)  {
             activate(output_state,init_value,event) } }
      { deactivate(state)
          state := EPC_inactive() }
      { exit(event,test_event)
         when (event = test_event) {
            exit := true } };

defconst  EPC_check_CS_other_staff_in_input  >> EPC_state_set =  (and, CS_no_other_staff);
defconst  EPC_check_CS_other_staff_in_output  >> EPC_state_set =  (and, CS_other_staff);
def_state EPC_check_CS_other_staff_in_active : EPC_state_set
          := EPC_check_CS_other_staff_in_input();
defconst  EPC_check_CS_other_staff_out_input  >> EPC_state_set =  (and, CS_other_staff);
defconst  EPC_check_CS_other_staff_out_output  >> EPC_state_set =  (and, CS_no_other_staff);
def_state EPC_check_CS_other_staff_out_active : EPC_state_set := EPC_inactive();

defconst  EPC_check_CS_operator_in_input  >> EPC_state_set =  (and, CS_no_operator);
defconst  EPC_check_CS_operator_in_output  >> EPC_state_set =  (and, CS_operator);
def_state EPC_check_CS_operator_in_active : EPC_state_set := EPC_check_CS_operator_in_input();
defconst  EPC_check_CS_operator_out_input  >> EPC_state_set =  (and, CS_operator);
defconst  EPC_check_CS_operator_out_output  >> EPC_state_set =  (and, CS_no_operator);
def_state EPC_check_CS_operator_out_active : EPC_state_set := EPC_inactive();

defconst  EPC_wait_for_open_input  >> EPC_state_set =  (and, Gate_closed);
defconst  EPC_wait_for_open_output  >> EPC_state_set =  (and, open_Gate);
def_state EPC_wait_for_open_active : EPC_state_set := EPC_wait_for_open_input();

defconst  EPC_wait_for_change_input  >> EPC_state_set =  (xor, open_Gate);
defconst  EPC_wait_for_change_output  >> EPC_state_set =  (xor, stop_Gate.Gate_open.close_Gate);
def_state EPC_wait_for_change_active : EPC_state_set := EPC_inactive();

defconst  EPC_wait_for_close_input  >> EPC_state_set =  (and, Gate_open);
defconst  EPC_wait_for_close_output  >> EPC_state_set =  (and, close_Gate);
def_state EPC_wait_for_close_active : EPC_state_set := EPC_inactive();

defconst  EPC_wait_for_command_input  >> EPC_state_set =  (and, stop_Gate);
defconst  EPC_wait_for_command_output  >> EPC_state_set =  (xor, close_Gate.open_Gate);
def_state EPC_wait_for_command_active : EPC_state_set := EPC_inactive();


defconst  EPC_wait_for_closed_input  >> EPC_state_set =  (and, close_Gate);
defconst  EPC_wait_for_closed_output  >> EPC_state_set =  (xor, Gate_closed.open_Gate);
def_state EPC_wait_for_closed_active : EPC_state_set := EPC_inactive();


def_trans_pattern EPC check_CS_other_staff_in
   (con,event,events)
   active(EPC_check_CS_other_staff_in_active, EPC_check_CS_other_staff_in_input()),
   (con,events) ? EPC_check_CS_other_staff_in_output(),
   event ? events,
   event = 'CS_other_staff',
   deactivate(EPC_check_CS_other_staff_in_active),
   activate(EPC_check_CS_other_staff_out_active,EPC_check_CS_other_staff_out_input(),
            'CS_other_staff');

def_trans_pattern EPC check_CS_other_staff_out
   (con,event,events)
   active(EPC_check_CS_other_staff_out_active, EPC_check_CS_other_staff_out_input()),
   (con,events) ? EPC_check_CS_other_staff_out_output(),
   event ? events,
   event = 'CS_no_other_staff',
   deactivate(EPC_check_CS_other_staff_out_active),
   activate(EPC_check_CS_other_staff_in_active,EPC_check_CS_other_staff_in_input(),
            'CS_no_other_staff');

def_trans_pattern EPC check_CS_operator_in
   (con,event,events)
   active(EPC_check_CS_operator_in_active, EPC_check_CS_operator_in_input()),
   (con,events) ? EPC_check_CS_operator_in_output(),
   event ? events,
   event = 'CS_operator',
   deactivate(EPC_check_CS_operator_in_active),
   activate(EPC_check_CS_operator_out_active,EPC_check_CS_operator_out_input(),
            'CS_operator');

def_trans_pattern EPC check_CS_operator_out
   (con,event,events)
   active(EPC_check_CS_operator_out_active, EPC_check_CS_operator_out_input()),
   (con,events) ? EPC_check_CS_operator_out_output(),
   event ? events,
   event = 'CS_no_operator',
   deactivate(EPC_check_CS_operator_out_active),
   activate(EPC_check_CS_operator_in_active,EPC_check_CS_operator_in_input(),
            'CS_no_operator');

def_trans_pattern EPC wait_for_open
   (con,event,events)
   active(EPC_wait_for_open_active, EPC_wait_for_open_input()),
   (con,events) ? EPC_wait_for_open_output(),
   event ? events,
   event = 'open_Gate',
   deactivate(EPC_wait_for_open_active),
   activate(EPC_wait_for_change_active,EPC_wait_for_change_input(),'open_Gate');

def_trans_pattern EPC wait_for_change
   (con,event,events)
   active(EPC_wait_for_change_active, EPC_wait_for_change_input()),
   (con,events) ? EPC_wait_for_change_output(),
   event ? events,
   deactivate(EPC_wait_for_change_active),
   activate_if(EPC_wait_for_command_active,EPC_wait_for_command_input(),event,'stop_Gate'),
   activate_if(EPC_wait_for_close_active,EPC_wait_for_close_input(),event,'Gate_open'),
   activate_if(EPC_wait_for_closed_active,EPC_wait_for_closed_input(),event,'close_Gate');

def_trans_pattern EPC wait_for_command
   (con,event,events)
   active(EPC_wait_for_command_active, EPC_wait_for_command_input()),
   (con,events) ? EPC_wait_for_command_output(),
   event ? events,
   deactivate(EPC_wait_for_command_active),
   activate_if(EPC_wait_for_change_active,EPC_wait_for_change_input(),event,'open_Gate'),
   activate_if(EPC_wait_for_closed_active,EPC_wait_for_closed_input(),event,'close_Gate');


def_trans_pattern EPC wait_for_close
   (con,event,events)
   active(EPC_wait_for_close_active, EPC_wait_for_close_input()),
   (con,events) ? EPC_wait_for_close_output(),
   event ? events,
   event = 'close_Gate',
   deactivate(EPC_wait_for_close_active),
   activate(EPC_wait_for_closed_active,EPC_wait_for_closed_input(),'close_Gate');


def_trans_pattern EPC wait_for_closed
```

```
  (con,event,events)
  active(EPC_wait_for_closed_active, EPC_wait_for_closed_input()),
  (con,events) ? EPC_wait_for_closed_output(),
  event ? events,
  deactivate(EPC_wait_for_closed_active),
  activate_if(EPC_wait_for_change_active,EPC_wait_for_change_input(),event,'open_Gate'),
  activate_if(EPC_wait_for_open_active,EPC_wait_for_open_input(),event,'Gate_closed'),
  activate_if(EPC_wait_for_command_active,EPC_wait_for_command_input(),event,'stop_Gate');
```

Listing 7.1: PL implementation of Dam EPCs

7.2 Implementation of Olympic Games Example

```
defset EPC_event_names = { User_prompted_for_login_details, Login_Data_Delivered,
                           Authenticate_user, Login_with_admin_privileges_Attempted,
                           User_Authenticated, Check_admin_connection_time,
                           Logon_Successful,  Admin_at_unusual_time, Admin_behaviour_ok,
                           Admin_at_unusual_location, Admin_behaviour_ok,
                           Event_logged_illegal_user,  Session_disconnected };

defconst  EPC_Introduce_user_password_input  >> EPC_state_set
          :=  (and, User_prompted_for_login_details);
defconst  EPC_Introduce_user_password_output  >> EPC_state_set
          :=  (and, Login_Data_Delivered);
def_state EPC_Introduce_user_password_active : EPC_state_set
          := EPC_Introduce_user_password_input();
defconst  EPC_Authenticate_user_input >> EPC_state_set
          := (and, Login_Data_Delivered);
defconst  EPC_Authenticate_user_output >> EPC_state_set
          := (xor, Login_with_admin_privileges_Attempted.User_Authenticated );
def_state EPC_Authenticate_user_active : EPC_state_set
          := EPC_inactive();
defconst  EPC_Check_admin_connection_time_input  >> EPC_state_set
          := (and, Login_with_admin_privileges_Attempted);
defconst  EPC_Check_admin_connection_time_output  >> EPC_state_set
          := (xor, Admin_at_unusual_time.Admin_behaviour_ok);
def_state EPC_Check_admin_connection_time_active :  EPC_state_set
          := EPC_inactive();
defconst  EPC_Create_new_session_input >> EPC_state_set
          := (and, User_Authenticated);
defconst  EPC_Create_new_session_output >> EPC_state_set
          := (and, Logon_Successful);
def_state EPC_Create_new_session_active :  EPC_state_set
          := EPC_inactive();
defconst  EPC_Log_irregular_admin_behaviour_pattern_input >> EPC_state_set :
          := (xor, Admin_at_unusual_time.Admin_at_unusual_location);
defconst  EPC_Log_irregular_admin_behaviour_pattern_output   >> EPC_state_set
          := (and, Event_logged_illegal_user);
def_state EPC_Log_irregular_admin_behaviour_pattern_active : EPC_state_set
```

```
          := EPC_inactive();
defconst  EPC_Check_admin_location_input >> EPC_state_set
          := (and, Admin_behaviour_ok);
defconst  EPC_Check_admin_location_output >> EPC_state_set
          := (xor, Admin_at_unusual_location.Admin_behaviour_ok);
def_state EPC_Check_admin_location_active :  EPC_state_set
          := EPC_inactive();
defconst  EPC_End_admin_login_session_input >> EPC_state_set
          := (and, Event_logged_illegal_user);
defconst  EPC_End_admin_login_session_output >> EPC_state_set
          := (and, Session_disconnected);
def_state EPC_End_admin_login_session_active :  EPC_state_set
          := EPC_inactive();


def_trans_pattern EPC introduce_user_password
   (con,event,events)
   active(EPC_Introduce_user_password_active, EPC_Introduce_user_password_input()),
   (con,events) ? EPC_Introduce_user_password_output(),
   event ? events,
   event = 'Login_Data_Delivered',
   deactivate(EPC_Introduce_user_password_active),
   activate(EPC_Authenticate_user_active,EPC_Authenticate_user_input(),
           'Login_Data_Delivered');


def_trans_pattern EPC authenticate_user
   (con,event,events)
   active(EPC_Authenticate_user_active, EPC_Authenticate_user_input()),
   (con,events) ? EPC_Authenticate_user_output(),
   event ? events,
   activate_if(EPC_Create_new_session_active,EPC_Create_new_session_input(),
               event,'User_Authenticated') ,
   activate_if(EPC_Check_admin_connection_time_active,
                EPC_Check_admin_connection_time_input(),
                event,'Login_with_admin_privileges_Attempted') ,
   deactivate(EPC_Authenticate_user_active);


def_trans_pattern EPC create_new_session
   (con,event,events)
   active(EPC_Create_new_session_active,EPC_Create_new_session_input()),
   (con,events) ? EPC_Create_new_session_output(),
   event ? events,
   event = 'Logon_Successful',
   deactivate(EPC_Create_new_session_active);


def_trans_pattern EPC check_admin_connection_time
   (con,event,events)
   active(EPC_Check_admin_connection_time_active,EPC_Check_admin_connection_time_input()),
   (con,events) ? EPC_Check_admin_connection_time_output(),
   event ? events,
   activate_if(EPC_Log_irregular_admin_behaviour_pattern_active,
               EPC_Log_irregular_admin_behaviour_pattern_input(),
               event,'Admin_at_unusual_time'),
    activate_if(EPC_Check_admin_location_active, EPC_Check_admin_location_input(),
                event,'Admin_behaviour_ok'),
    deactivate(EPC_Check_admin_connection_time_active);


def_trans_pattern EPC check_admin_location
    (con,event,events)
    active(EPC_Check_admin_location_active,EPC_Check_admin_location_input()),
    (con,events) ? EPC_Check_admin_location_output(),
    event ? events,
    activate_if(EPC_Log_irregular_admin_behaviour_pattern_active,
                EPC_Log_irregular_admin_behaviour_pattern_input(),
                event,'Admin_at_unusual_location'),
    exit(event,'Admin_behaviour_ok'),
    deactivate(EPC_Check_admin_location_active);


def_trans_pattern EPC log_irregular_admin_behaviour_pattern
    (con,event,events)
    active(EPC_Log_irregular_admin_behaviour_pattern_active,
           EPC_Log_irregular_admin_behaviour_pattern_input()),
    (con,events) ? EPC_Log_irregular_admin_behaviour_pattern_output(),
    event ? events,
    event = 'Event_logged_illegal_user',
    deactivate(EPC_Log_irregular_admin_behaviour_pattern_active),
    activate(EPC_End_admin_login_session_active,EPC_End_admin_login_session_input(),event);

def_trans_pattern EPC end_admin_login_session
    (con,event,events)
    active(EPC_End_admin_login_session_active,EPC_End_admin_login_session_input()),
    (con,events) ? EPC_End_admin_login_session_output(),
    event ? events,
    deactivate(EPC_End_admin_login_session_active),
    exit(event,'Session_disconnected');
```

Listing 7.2: PL implementation of Olympic Games EPC