From Declarative Signatures to Misuse IDS

Jean-Philippe Pouzol and Mireille Ducassé

IRISA/INSA de Rennes - Campus Universitaire de Beaulieu
35042 Rennes Cedex, France ; {pouzol,ducasse}@irisa.fr

Abstract In many existing misuse intrusion detection systems, intrusion signatures are very close to the detection algorithms. As a consequence, they contain too many cumbersome details. Recent work have proposed declarative signature languages that raise the level of abstraction when writing signatures. However, these languages do not always come with operational support. In this article, we show how to transform such declarative signatures into operational ones. This process points out several technical details which must be considered with care when performing the translation by hand, but which can be systematically handled.

A signature specification language named *Sutekh* is proposed. Its declarative semantics is precisely described. To produce rules for existing rule-based IDS from *Sutekh* signatures, an algorithm, based on the construction of a state-transition diagram, is given.

1 Introduction

Many formalisms and algorithms have been proposed to specify intrusion signatures in order to detect misuses. Among the intrusion detection systems (IDS) briefly described by Axelsson in [1], one can cite state-transition diagrams in STAT [8], colored Petri nets in IDIOT [10], condition-action rules in ASAX [7], expert systems in EMERALD [15]. The languages proposed in these articles are strongly dedicated to the underlying search algorithm. As a consequence, the signatures are cluttered with many operational details which make them hard to specify and maintain.

More recent approaches propose to specify signatures of intrusions in a declarative flavor, for example MuSigs defined by Lin et al. [11] and LaDAA defined by Gérard [6]. LAMBDA, defined by Cuppens and Ortalo [4], and ADeLe, proposed by Michel and Mé [13], are languages dedicated to describe both attacks and signatures. The aim of all these high-level languages is to specify signatures, regardless of any detection process, by describing relations between events in an audit trail. However, either these languages ignore functionalities provided by others, or they do not offer any operational support.

This article firstly proposes Sutekh, a declarative signature language providing a combination of functionalities at least as complete as the union of what is offered by other declarative systems. In addition, we show how to produce operational detection rules from declarative signatures.

Sutekh provides sequence, alternative, partial order, negation, event correlation by logical variables, condition verification and alert triggering. These functionalities are formally defined by a declarative semantics that rigorously describes which sequences of events in an audit trail are considered as occurrences of a signature. This semantics extends earlier work of Chakravarthy [3] in the context of active databases.

In order to implement the signature detection, we propose to produce rules for rule-based IDS from declarative signatures. We first define an algorithm to build a graph structure (called *Signature Graph* or shortly *SigGraph*) representing the evolution of the detection process. From this graph, rules for different target IDS can be automatically generated as illustrated by two examples involving ASAX and the expert system P-BEST used in EMERALD.

Our *SigGraph* is a state-transition diagram close to the one used in STAT. Its systematic construction, however, exhibits a number of technical problems in particular with the combination of logical variables, partial order and negation. With our approach, these problems are solved once for all.

The main contribution of this article is, therefore, to exhibit a whole chain of transformations from declarative signature specifications to operational rules. When specifying or maintaining a signature base, one can thus concentrate on the essential aspects of signatures captured by the declarative semantics. The operational aspects will be automatically addressed.

The remainder of this paper is structured as follows. Section 2 presents Sutekh, illustrated by the complete specification of a signature. Section 3 gives the declarative semantics of Sutekh. Section 4 details the algorithm we propose to produce rules for rule-based IDS. Related work is discussed in section 5, and the paper is concluded in section 6.

2 Overview of Sutekh: a Declarative Signature Specification Language

This section presents the syntax and the informal semantics of Sutekh, a declarative signature specification language. We first introduce basic elements called *filters*, and then show how to combine filters to build signatures. Finally, we propose a complete example of specification. The formal semantics is given in Sec. 3.

2.1 Basic Blocks: Filters

Filters are "single event" signatures. They are used to define constraints on an event. Figure 1 shows an abstract syntax of filters. Each constraint specifies a relation between an event field and a term. A term can be either a set (in fact, a disjunction) of constant values or a variable. If an event satisfies all the constraints, this event matches the filter. Four constraint operators are defined, equality (`=`), inequality (`!=`), bitwise-and matching (`&=`), and regular expression matching (`#=`). This list could be extended if necessary.

```
< Filter >       ::=  < Constraint >+
< Constraint >   ::=  < EventField >  < Op >  < Term >
< Op >           ::=  = | != | &= | #=
< Term >         ::=  < ConstantValue >+ | < VariableName >
```

Figure1. Abstract syntax of filters in Sutekh

```
< Signature >      ::=  < SignatureName > ( < ArgName >* ) < Body > .
< Body >           ::=  < Filter >
                     |  < Body > (then | or | and | if_not ) < Body >
                     |  < Body >  such_that  < ExternalPred >
                     |  < Body >  trigger  < ExternalProc >
                     |  < SubSignature >
                     |  ( < Body > )
< ExternalPred >   ::=  < PredName > ( < Term >* )
< SubSignature >   ::=  < SignatureName > ( < Term >* )
```

Figure2. Abstract syntax of signatures in Sutekh

2.2 Signature Specification in Sutekh

Figure 2 shows an abstract syntax of signatures in Sutekh. Signatures are declared within $<$ *Signature* $>$ statements which are similar to function declaration in traditional programming languages.

Signatures are built in a compositional way as shown by the rule:

$<$ *Boby* $>$ ::= $<$ *Body* $>$ (**then** | **or** | **and** | **if_not**) $<$ *Body* $>$

- **Sequence** ($\mathbf{S = S_1}$ **then** $\mathbf{S_2}$): an instance of S is found if an instance of S_1 is found and if an instance of S_2 is found in the remaining part of the trail.
- **Choice** ($\mathbf{S = S_1}$ **or** $\mathbf{S_2}$): an instance of S is found if either an instance of S_1 or an instance of S_2 is found.
- **Interleaving** ($\mathbf{S = S_1}$ **and** $\mathbf{S_2}$): an instance of S is found if instances of S_1 and S_2 can be independently found in the trail. This describes a partial order on events which compose the instance.
- **Negation** ($\mathbf{S = S_1}$ **if_not** $\mathbf{S_2}$): an instance of S is found if an instance of S_1 is found, and if there exists no instance of S_2 between the first and the last events of the instance of S_1.

The base case of this inductive definition is a signature composed of one filter. An instance of a single-filter signature is found if an event matches the filter as described in Sec. 2.1.

The **such_that** statement is used to call external logical predicates. An instance of signature S = (S_1 such_that P) is found if an instance of S is found and if predicate P holds.

The **trigger** statement is used to call an external procedure, for example, to report an alert when an instance of a signature is found. Since the expression (S trigger A) is a signature, it can be used as a component of another signature: this provides a means to raise "partial" alerts before the attack is completed.

2.3 An Example of Signature in Sutekh

This section presents an example of a signature to detect an attack exploiting a buffer overflow in the `admintool` program distributed in the Solaris environment. This vulnerability has been reported in the CERT advisory CA-1996-16 [2]. `admintool` is a graphical front end to several administration tasks, in particular to install new software packages through the `pkgadd` command. All Solaris packages contain two files named `pkginfo` and `pkgmap`. The first step to overflow `admintool` is to create these two files in a directory (`/tmp/EXP` in this example) and put an overflowing string in them. The second step is to execute `admintool` and install the fake package located in `/tmp/EXP`.

Figure 3 presents a signature of this exploit in a BSM [16] audit trail. After the declaration of several constant values corresponding to event identifiers in BSM, we define two filters:

`createFileLike(Regexp, File, UserId)`: a filter for events corresponding to file creation (`AUE_CREAT` and `AUE_OPEN_WC`)[1]. The name of the created file must match the regexp in parameter `Regexp` and is unified with the parameter `File`. The real user ID is unified with the parameter `UserId`.

`execProg(ProgName, UserId)`: a filter for events corresponding to program execution (`AUE_EXEC` and `AUE_EXECVE`). The name of the executed program is unified with the parameter `ProgName`, and the real user ID is unified with the parameter `UserId`.

These two filters are used in the last signature, `admintool()`. Since the two files can be created in an arbitrary order, the **and** operator connects the two `createFileLike` filters. The use of the `User` variable as parameter of these filters specifies that the files must be created by the same user[2]. The names of the two files are respectively bound to variables `File1` and `File2`.

We add a constraint, using the **such_that** statement and the external predicate `same_directory`, which verifies that the names of the two files have the same prefix (ie. the files are located in the same directory).

Using the **then** statement, we specify that the creation of the two files must be followed by the execution of `admintool`. The variable `User` specifies again that this operation must be performed by the very user who created the files.

[1] In fact, at least eight system calls can create a file on Solaris systems. Adding the six missing identifiers presents no difficulty.

[2] This attack can also be performed by several users. In such a case, the signature would be a variant of the one shown here.

```
% External predicate used   : same_directory
% External procedure called : admintool_alert
% ----------------------------------------------------------------------
% Declaration of constants
const AUE_CREAT   =  4. % eventID for syscall 'create'
const AUE_OPEN_WC = 77. % eventID for syscall 'open' in write/create mode
const AUE_EXEC    =  7. % eventID for syscall 'exec'
const AUE_EXECVE  = 23. % eventID for syscall 'execve'
% ----------------------------------------------------------------------
% Filter for file creation
createFileLike(Regexp, File, UserId)
  = [ eventID      = ( AUE_CREAT | AUE_OPEN_WC ),
      realUserID   = UserId,
      path        #= Regexp,
      path         = File ].

% Filter for program execution
execProg(ProgName, UserId)
  = [ eventID      = ( AUE_EXEC | AUE_EXECVE ),
      path         = ProgName,
      realUserId   = UserId ].
% ----------------------------------------------------------------------
% Signature for the admintool exploit
admintool()
  = ( ( ( createFileLike("*/pkginfo", File1, User)
          and
          createFileLike("*/pkgmap", File2, User) )
        such_that same_directory(File1, File2) )
      then
      execProg("/usr/bin/admintool", User) )
    trigger admintool_alert(User).
```

Figure3. Signature in Sutekh of an exploit using `admintool`

Finally, the **trigger** statement specifies the external procedure used to report the alert (not detailed here).

3 Semantics of Sutekh

This section presents the declarative semantics of signatures in Sutekh. We formally present what we call *event*, *filter* and *event matching*. Then, the semantics is inductively defined on the syntactic constructs proposed in the previous section.

3.1 Event Matching

An event is a structure where fields are named. Let $\mathcal{L}$ be the set of event field names and $\mathcal{D}$ the domain of values of these fields.

Definition 1 (Event). *An event is a set of pairs* (l, v) *where* $l \in \mathcal{L}$ *and* $v \in \mathcal{D}$. *We note* $l(\mathrm{E})$ *the value* v *of field* l *in event* E.

Definition 2 (Filter). *Given a set of variable* $\mathcal{V}$, *a filter* F *is a set of constraints* $\{\mathrm{C_1}, ..., \mathrm{C_n}\}$. *Each* $\mathrm{C_k}$ *has the form* $(\mathrm{l_k}, \mathrm{r_k}, \mathrm{t_k})$ *where* $\mathrm{l_k} \in \mathcal{L}$ *is an event field,* $\mathrm{r_k}$ *is a binary relation (eg. equality), and* $\mathrm{t_k} \in (\mathcal{V} \cup \mathcal{P}(\mathcal{D}))$ *is a variable or a set of constants (we note* $\mathcal{P}(\mathcal{D})$ *the power set of* $\mathcal{D}$*).*

Definition 3 (Substitution). *Given a set of variables* $\mathcal{V} = \{V_1, \ldots, V_n\}$, *a substitution* σ *is a total function from* $\mathcal{V}$ *to* $\mathcal{D}$.

Definition 4 (Constraint Satisfaction). *We say that an event* E *satisfies a constraint* $(\mathrm{l}, \mathrm{r}, \mathrm{t})$ *with a substitution* σ *iff :*

$$(t \in \mathcal{V} \;\Rightarrow\; (l(E) \; r \; \sigma(t))) \quad \wedge \quad (t \in \mathcal{P}(\mathcal{D}) \;\Rightarrow\; \exists x \in t \; . \; (l(E) \; r \; x))$$

Definition 5 (Event Matching). *We say that an event* E *satisfies a filter* F *with a substitution* σ, *and we note* $(\mathrm{E}\, \Theta_\sigma\, \mathrm{F})$, *if and only if all constraints in* F *are satisfied by* E. *We also say that "event* E *matches filter* F *with substitution* σ*".*

3.2 Instance of a Signature

Definition 6 (Trail). *A trail is a totally ordered sequence of events. The* i^{th} *element of a trail* T *is noted* T[i]. *We call* i *the "position" of event* T[i].

It is important to distinguish the position of an event from the time information it contains. Since signatures express relations between events in the trail, the semantics of signatures only takes into account the position of the events. However, constraints on time information can be declared in **such_that** statements.

Definition 7 (Instance of a Signature). *Let* T *be a trail,* S *a signature and* σ *a substitution of the variables of* S. *We characterize an instance of the signature* S, *in trail* T, *between two positions* i_α *and* i_β, *with the substitution* σ, *by means of the relation:* $\boldsymbol{T, \sigma, (i_\alpha, i_\beta) \models S}$.

Figure 4 inductively defines the $\models$ relation on each Sutekh syntactic construct. It is important to note that the bounds i_α and i_β are respectively the position of the first and the last event of the instance.

The **Filter** definition ensures that the bounds coincide and that the event occurring at this position matches the filter F with substitution σ.

The **then** definition ensures that instances of S_1 and S_2 occur and that the ending position of the first part of the sequence is strictly lower than the starting position of the second part.

The **or** definition ensures that at least one of the two parts of the choice occurs.

The **and** definition ensures that the two parts of the interleaving occur and that the starting position (resp. ending position) of the whole instance is the minimum (resp. maximum) of the starting positions (resp. ending positions) of the two parts.

The **if_not** definition ensures that an instance of S_1 occurs, and that no instance of S_2 occurs between starting and ending positions of the instance of S_1. The bounds of the search interval of S_2 are strict ($i_\alpha < i_1$ and $i_2 < i_\beta$). The lower bound is strict because, from an operational point of view, one only wants to search for S_2 if the beginning of an instance of S_1 is detected. The upper bound is strict because if an event satisfies both the last filters of S_1 and S_2, an occurrence of S_1 exists, and an alarm must be raised.

The **such_that** definition ensures that an instance of S occurs and that $\sigma(Pred)$ holds. We note $\sigma(Pred)$ the result of applying the substitution σ to the variables in the predicate $Pred$.

The **trigger** definition merely ensures that an instance of S occurs. Indeed, this statement is used to produce side effects that *do not* interfered with the detection process (eg. raising an alarm, producing debugging information).

4 Transforming Sutekh Signatures into Rules

The previous section detailed the declarative semantics of Sutekh. This semantics, intended for the end-users, gives a precise meaning to signatures without introducing operational considerations. This is motivated by the fact that end-users do not need to know how the search will be performed. As a consequence, the transformation of declarative signatures into operational ones must be done automatically. Therefore, we propose to compile *Sutekh* signatures into existing misuse IDS formalisms and focus on rule-based systems. This section proposes an algorithm to achieve this task.

The algorithm builds, as an intermediate representation, a state-transition diagram called a *SigGraph*. In the following, we first present the different components of this graph, and then show how operational rules can be generated from this structure. The remainder of the section then details the three main steps of the *SigGraph* construction.

4.1 *SigGraph* Definition

A *SigGraph* is a kind of non-deterministic finite automaton. Each node corresponds to a state in the process of searching an audit trail for signatures. Arcs are oriented and labeled with Sutekh filters. Outgoing transitions from a node correspond to events the IDS reacts upon when it is in this state. Transitions may be labeled with predicates declared in **such_that** statements. Intuitively, this means that the transition can be made if the event satisfies both the constraints of the filter and of the predicates. Procedures can also be attached to transitions: they are called by the IDS when the transitions are performed.

$$
\begin{array}{lll}
T,\sigma,(i_\alpha,i_\beta) \models \textbf{Filter}\, F & \text{iff} & (i_\alpha = i_\beta) \;\wedge\; (T[i_\alpha]\,\Theta_\sigma\, F) \\[1em]
T,\sigma,(i_\alpha,i_\beta) \models S_1\,\textbf{then}\, S_2 & \text{iff} & \exists(i_1,i_2).(\\
 & & \qquad\quad T,\sigma,(i_\alpha,i_1) \models S_1 \\
 & & \qquad \wedge\; T,\sigma,(i_2,i_\beta) \models S_2 \\
 & & \qquad \wedge\; (i_\alpha \le i_1 < i_2 \le i_\beta)\;) \\[1em]
T,\sigma,(i_\alpha,i_\beta) \models S_1\,\textbf{or}\, S_2 & \text{iff} & \quad T,\sigma,(i_\alpha,i_\beta) \models S_1 \\
 & & \vee\; T,\sigma,(i_\alpha,i_\beta) \models S_2 \\[1em]
T,\sigma,(i_\alpha,i_\beta) \models S_1\,\textbf{and}\, S_2 & \text{iff} & \exists(i_1,i_2,i_3,i_4).(\\
 & & \qquad\quad T,\sigma,(i_1,i_2) \models S_1 \\
 & & \qquad \wedge\; T,\sigma,(i_3,i_4) \models S_2 \\
 & & \qquad \wedge\; i_\alpha = min(i_1,i_3) \\
 & & \qquad \wedge\; i_\beta = max(i_2,i_4)\;) \\
T,\sigma,(i_\alpha,i_\beta) \models S_1\,\textbf{if\_not}\, S_2 & \text{iff} & \quad T,\sigma,(i_\alpha,i_\beta) \models S_1 \\
 & & \wedge\; \forall(i_1,i_2).(\\
 & & \qquad (i_\alpha < i_1 \le i_2 < i_\beta) \\
 & & \qquad\quad \Rightarrow T,\sigma,(i_\alpha,i_\beta) \not\models S2\;) \\[1em]
T,\sigma,(i_\alpha,i_\beta) \models S\,\textbf{such\_that}\, P & \text{iff} & T,\sigma,(i_\alpha,i_\beta) \models S \;\wedge\; \sigma(P) \\
T,\sigma,(i_\alpha,i_\beta) \models S\,\textbf{trigger}\, P & \text{iff} & T,\sigma,(i_\alpha,i_\beta) \models S
\end{array}
$$

Figure4. Semantics of signatures in Sutekh

Given a signature S we write $Var(S)$ for the set of all variables used in S, $Pred(S)$ the set of predicates used in **such_that** statements and $Proc(S)$ the set of procedures used in **trigger** statements. We also use the standard notation $\mathcal{P}(E)$ for the power set of a set E.

We give a formal definition of a *SigGraph* graph, and Fig. 5 illustrates this definition with an excerpt of such a graph.

Definition 8 (SigGraph). *Given a signature S, SigGraph(S) is defined as a 4-tuple* $(\Gamma, Init, F, T)$ *where:*

- Γ *is a set of nodes,* $\Gamma \subset (Id \times \mathcal{P}(Var(S)) \times \{rec \,|\, nonrec\})$
- $Init$ *is the initial state of the graph,* $Init \in \Gamma$
- $Final$ *is the set of (labeled) final states,* $Final \subset (\Gamma \times \{accept \,|\, reject\})$
- T *is the transition relation,* $T \subset (\Gamma \times \mathcal{A} \times \Gamma)$ *where*
  $\mathcal{A}$ *is the set of transitions,*
  $\mathcal{A} \subset (Filter \times \mathcal{P}(Pred(S)) \times \mathcal{P}(Proc(S)) \times \{forward \,|\, backward\})$

Nodes. They are identified by an identifier Id. Each node contains a set, named $BoundVars$, of variables that have been assigned a value before reaching this state. Nodes are also labeled by "*rec*" or "*nonrec*". This flag indicates if the state is *recurrent* or not. A state is said to be *recurrent* if, once it has been reached, the IDS must be ready to transit again from it (possibly using another transition). In Fig. 5, node **n2** is "*nonrec*" and node **n3** is "*rec*". Intuitively, **n3** is a choice point (two arrows start from it), while **n2** is not. How to determine if a node is recurrent or not is detailed in Sec. 4.5. This distinction between recurrent and not recurrent states is the same as the one made between *consuming* and *nonconsuming* rules in STATL [5]. The existence of recurrent states in a *SigGraph* can also be compared to the duplication of tokens in IDIOT Petri nets [9].

Final states. A *SigGraph* actually contains only one final state. However, due to the existence of temporary final states during the construction of the graph, $Final$ is defined as a *subset* of Γ. Final states are labeled by a flag "*accept*" or "*reject*". This flag is used to discriminate temporary final states during the construction; their precise meaning will be detailed in Sec. 4.3. Figure 5 does not contain any final state.

Transitions. Transitions are labeled by a filter, a set $SuchPred$ of predicates declared in **such_that** statements ($SuchPred \in \mathcal{P}(Pred(S))$) and a set $ProcTrig$ of procedures declared in **trigger** statements ($ProcTrig \in \mathcal{P}(Proc(S))$). Given a transition, the predicates in $SuchPred$ must hold to perform the transition; the procedures in $ProcTrig$ are called when the transition is made.

Labels "*forward*" and "*backward*" are used to discriminate transition types in the graph. Intuitively, "*forward*" arrows correspond to a progress in the discovery of an instance. "*Backward*" arrows correspond to a backtracking to a previous state in the graph because an instance of a negative part of an **if_not** is found. Figure 5 contains two "*forward*" transitions from node **n1**. They respectively correspond to the detection of a link creation or a file reading by the same process as the one bound to variable **P** in node **n1**. Figure 5 also contains one "*backward*" transition which cancels the search if process **P** terminates.

4.2 Generating Rules from a *SigGraph*

This section sketches, on two examples, how a *SigGraph* can be used to generate rules for IDS. The two examples respectively use the rule-based languages of ASAX and P-BEST.

Generating Russel Rules. Russel is the rule-based language of the ASAX system. An overview of this language can be found in [7], a more complete definition of the language with its semantics is proposed in [14].

Briefly, a Russel program is a collection of rules where each rule consists of several "Condition → Actions" statements. These rules can be parameterized by formal parameters. During the detection, ASAX handles a set of instances of these rules (ie. where formal parameters have been substituted by effective

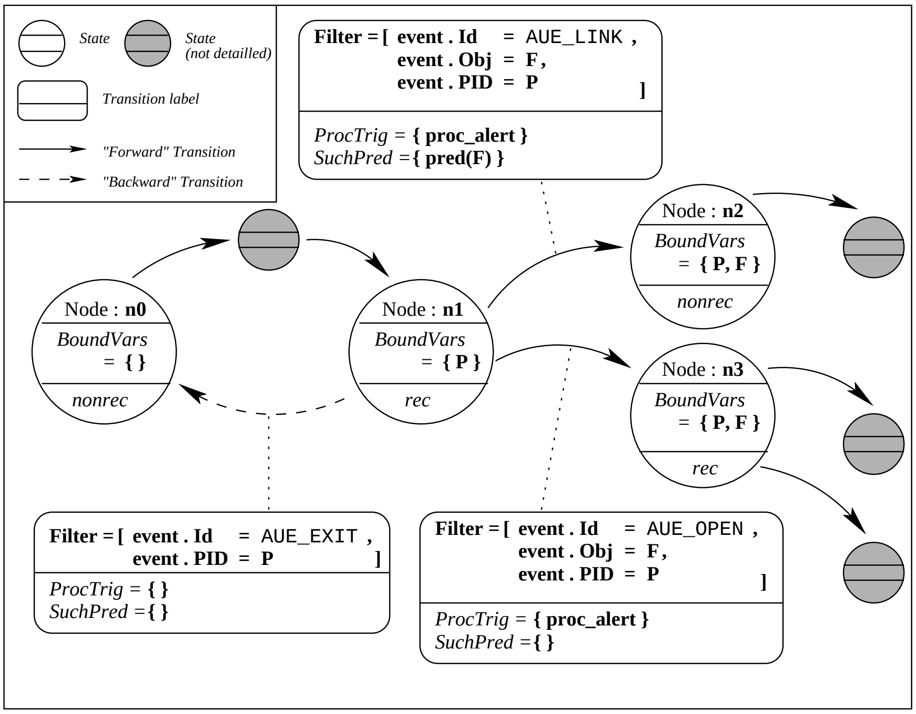

Figure5. Excerpt of a *SigGraph* representing a Sutekh signature

```
rule r1(int P);
begin
 if
   ( (event.ID=AUE_LINK) and (event.PID=P) and pred(event.Obj)=true) )
    --> begin
         trigger off for_next r2(P,event.Obj);
         trigger off for_next r1(P);
         proc_alert
        end;

   ( (event.ID=AUE_OPEN) and (event.PID=P) )
    --> begin
         trigger off for_next r3(P,event.Obj);
         trigger off for_next r1(P);
         proc_alert
        end;

   ( (event.ID=AUE_EXIT) and (event.PID=P) )
    --> trigger off for_next r0;

   true
     --> trigger off for_next r1(P)
 fi
end;
```

Figure6. Russel rule generated from node **n1** of Fig. 5

ones) called the "set of active rules". Each time an event is generated by the audit, it is processed by each active rule. If the event satisfies one "condition" in a rule, the corresponding "action" is executed. The "action" part of a rule contains instructions which modify the set of active rules used to process the next event.

Each node in the graph is transformed into a Russel rule. Figure 6 shows the rule corresponding to node **n1** of Fig. 5. The $BoundVars$ set of the node gives the number and the name of parameters of the rule. Rule `r1` has one parameter, `P`. All outgoing transitions from the node correspond to a "Condition $\rightarrow$ Action" statement in the rule.

The condition part of the statement is obtained by conjunction of (a) constraints in the filter which used bound variables or constant values, and (b) predicates in $SuchPred$. Bound variables are replaced by the corresponding parameters in the rule. Unbound variables are replaced by the audit field they are unified with. For example, in the first part of rule `r1`, variable **P** is bound, then `event.PID` is compared to parameter `P`; variable **F** is not bound, then predicate **pred** is evaluated with `event.Obj`.

The action part of the statement is obtained by (a) adding in the set of active rules for next event an instance of the rule corresponding to the successor of node **n1** through this transition (the `trigger off` statements in the example), and (b) calling procedures in $ProcTrig$.

Finally, since the node is labeled "*rec*", except if an event `AUE_EXIT` is detected, the rule always adds itself to the set of active rules for next event.

Generating P-Best Rules. Production-Based Expert System Toolset (P-BEST) is a an expert system used, among others, in the EMERALD intrusion detection system [15]. A description of P-BEST capabilities can be found in [12].

A P-BEST program consists of the definition of several fact types, and a set of inference rules on these facts. Inference rules are composed of two parts: the first part is a guard which tests the existence of facts satisfying logical expressions; the second part is composed of actions upon the fact base (adding, removing, modifying facts) and of calls to external functions.

Figure 7 shows the rule corresponding to node **n1** of Fig. 5. One fact type is created for each node in the graph. This is achieved by the `ptype` statements. Values stored in facts correspond to bound variables in the node.

An inference rule is created for each transition in the graph. The guard of a rule tests the existence of a fact corresponding to the origin of the transition. In Fig. 7 all three rules test the existence of a fact `x` of type `state1` (in the `[+x: state1]` statement). Then, rules test if an event fact `e` satisfies the condition in the `[?|`< *condition* >`]` statements. The generation of the condition is similar to what has been done for the generation of Russel rules; here, bound variables are substituted by values stored in the fact `x`.

If the event satisfies the condition, (a) the fact corresponding to the successor of node **n1** is created (for example, in rule `r1a`, a fact `state2` is created in the `[+state2 ...]` statement), and (b) the procedures in the $ProgTrig$ set are called (`[!|`< *call* >`]` statement).

```
/* Declaration of one fact type per node */

ptype[ state0 ]
ptype[ state1 p:int]
ptype[ state2 p:int,
              f:string]
ptype[ state3 p:int,
              f:string]

/* Definition of one rule per transition */

rule[r1a]
 [+x: state1]
 [+e: event]
 [? | e.ID == 'AUE_LINK.
      e.PID == x.P,
      'pred(e.Obj) == 'True]
==>
 [+state2 | p = x.P,
            f = e.Obj]
 [! | proc_alert]

rule[r1b]
 [+x: state1]
 [+e: event]
 [? | e.ID == 'AUE_OPEN.
      e.PID == x.P]

==>
 [+state3 | p = x.P,
            f = e.Obj]
 [! | proc_alert]

rule[r1c]
 [+x: state1]
 [+e: event]
 [? | e.ID == 'AUE_EXIT.
      e.PID == x.P,

==>
 [+state0]
 [-| x]
```

Figure7. P-BEST facts and rules generated from node **n1** in Fig. 5

If the node is labeled "*rec*", the fact corresponding to the state is not removed from the fact base, except if the transition is "*backward*". In the example, rule `r1c` corresponds to a "*backward*″ transition, then fact `x` is removed (`[-| x]` statement). If the node is labeled "*nonrec*", the fact is always removed (not shown in the example).

4.3 Building the Core Structure of a *SigGraph*

Now that we know what can be done with the graph, let us go back to the algorithm to build it. The first step in the construction of the graph consists of creating its core structure (ie. the set of node Γ and the transition relation T).

We define a function $\mathcal{G}$ which builds the structure of the graph from Sutekh signatures. $\mathcal{G}$ is inductively defined on the syntactic constructs of Sutekh. We present a definition of this function for each construct, and illustrate it on an example in Fig 8. In this figure, filters are only represented by letters (A_1, A_2, B, C, D) since constraints in filters are not used in the definition of $\mathcal{G}$.

Filter. The graph for a filter has two nodes, the first one is $Init$, the second one belongs to $Final$ and is marked "*accept*". $BoundVars$ and $ProcTrig$ of those two nodes are empty sets. Each node is marked *nonrec* (this is a default value which will possibly be modified later in the algorithm). The transition between these nodes is labeled by the filter and marked "$forward$". An example of such a graph is depicted in Fig. (8a) with the filter **B** in the right-hand side of the **and**.

Choice ($\mathbf{S} = \mathbf{S_1}$ or $\mathbf{S_2}$). $\mathcal{G}(S)$ is obtained by merging $Init$ states and merging final states of the two graphs. This operation is illustrated in Fig. (8a) by the graph corresponding to (A_1 **or** A_2) in the left-hand side of the **and**.

Interleaving ($\mathbf{S} = \mathbf{S_1}$ and $\mathbf{S_2}$). $\mathcal{G}(S)$ is obtained by performing the cartesian product of the two graphs. The *Init* state of $\mathcal{G}(S)$ is the one corresponding to $(Init(\mathcal{G}(S_1)),\ Init(\mathcal{G}(S_2)))$; the final state of $\mathcal{G}(S)$ is the one corresponding to $(Final(\mathcal{G}(S_1)),\ Final(\mathcal{G}(S_2)))$. This operation is illustrated in Fig. (8a) by the graph corresponding to $((A_1\ \mathbf{or}\ A_2)\ \mathbf{and}\ B)$.

Sequence ($\mathbf{S} = \mathbf{S_1}$ then $\mathbf{S_2}$). The *Init* state of the $\mathcal{G}(S)$ is the *Init* state of $\mathcal{G}(S_1)$; its final state is the final state of $\mathcal{G}(S_2)$. The *Init* state of $\mathcal{G}(S_2)$ is merged with the final state of $\mathcal{G}(S_1)$. This operation is illustrated in the top of Fig. 8.b by the graph corresponding to $(C\ \mathbf{then}\ D)$, in the right-hand side of the **if_not**.

Negation ($\mathbf{S} = \mathbf{S_1}$ if_not $\mathbf{S_2}$). Transforming an **if_not** is like transforming an **and**: the IDS must look in parallel for an instance of the positive part *and* an instance of the negative part of the **if_not**. However, if an instance of the negative part is found, the whole search must be cancelled.
The transformation of **if_not** is split into four steps:

- **Step 1**: the cartesian product of $\mathcal{G}(S_1)$ and $\mathcal{G}(S_2)$ is computed. States resulting from the product of either $Final(\mathcal{G}(S_1))$ or $Final(\mathcal{G}(S_2))$ are final states of $\mathcal{G}(S)$. Final states resulting from $Final(\mathcal{G}(S_1))$ are marked "*accept*", those resulting from $Final(\mathcal{G}(S_2))$ are marked "*reject*". The node $(Final(\mathcal{G}(S_1)),\ Final(\mathcal{G}(S_2)))$ is marked "*accept*". Indeed, in this case, even if an instance of the negative part of the signature is found, an instance of the positive one is found too. Labeling such states with "*accept*" is not an arbitrary decision, it means that recognition has priority over cancellation. This is consistent with the semantics of Fig. 4.
- **Step 2**: Since we are only interested in instances of S_2 after the first event of the instance of S_1 (see semantics, Fig. 4), all new transitions from the *Init* state which were not present in $\mathcal{G}(S_1)$ are suppressed. This makes some states unreachable: they are suppressed in turn. In Fig (8b), this operation removes one transition from *Init*, labeled by C, and leads to the suppression of two nodes (and eight transitions attached to them).
- **Step 3**: All transitions after a final state are suppressed. As in previous steps, unreachable states are suppressed. In Fig (8b), this operation leads to the suppression of one node and five transitions.
- **Step 4**: (a) all transitions to final states marked "*reject*" are replaced by "*backward*" transitions to the *Init* state, and (b) all final states marked "*accept*" are merged in a unique final state marked "*accept*".

such_that ($\mathbf{S} = \mathbf{S_1}$ such_that **pred**). $\mathcal{G}(S)$ is obtained by adding, in all transitions $\mathcal{G}(S_1)$, the predicate *pred* to *SuchPred*. This will be refined in the following.

trigger ($\mathbf{S} = \mathbf{S_1}$ trigger **proc**). $\mathcal{G}(S)$ is obtained by adding, in all transitions to final states of $\mathcal{G}(S_1)$, the procedure *proc* to *ProcTrig*.

4.4 Static Analysis of Variable Binding in a *SigGraph*

This section details how to compute the *BoundVars* set for each node of the graph. It is necessary to compute these sets because in a rule-based IDS, the first

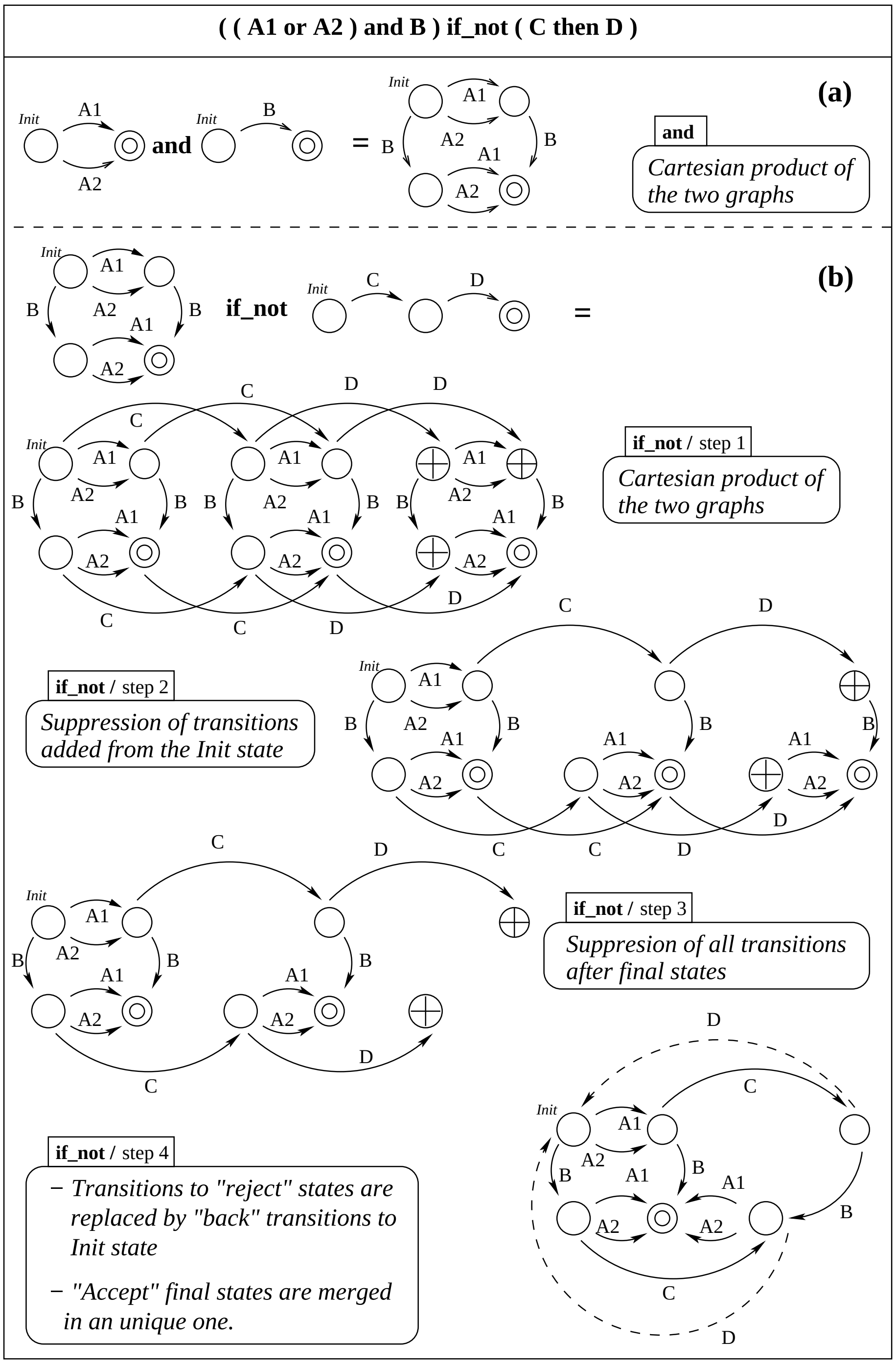

Figure8. Building the core structure of the graph from Sutekh specifications

time a variable is encountered, it takes the value of the corresponding event field in the constraint; in all other constraints, using the variable generates a boolean test between the value stored in the variable and the event field.

Due to disjunctions in signatures, several paths may lead to the same node in the graph. On these different paths, filters attached to transitions may not instantiate the same variables, and then, the set of bound variables in a node may differ according to the path which led to the node.

The algorithm proposed in this section performs two tasks. Firstly, it computes the $BoundVars$ set for each node. Secondly, each time it discovers that several $BoundVars$ sets are possible according to the path that led to the node, it duplicates the node. After this duplication, for each node, the $BoundVars$ set is unique for each path leading to it.

Given a node n and a transition t, which connects node n′ to node n, we define the set VarFrom(n, t) as:
$VarFrom(n,t) = BoundVars(n') \cup \{\text{Variables in equality constraints in } t\}$.
Namely, when transiting from one state to another, the bound variables in the destination states are those already bound in addition to those unified when matching the filter attached to the transition.

Figure 9 shows an execution of the algorithm detailed later on in this section. In this figure, filters are abstracted by their name and variables used in equality constraints. In the "initial graph" of Fig. 9 the $VarFrom$ set related to node **n2** and transition $\mathbf{C}(\mathbf{X},\mathbf{Y})$ is:

$$VarFrom(\mathbf{n2}, \mathbf{C}(\mathbf{X},\mathbf{Y})) = \{\} \cup \{\mathbf{X},\mathbf{Y}\} = \{\mathbf{X},\mathbf{Y}\}$$

Definition 9. *Given a node n, we define the relation of equivalence $\mathcal{R}_n$ over transitions with end connection in node n as:*

$$t_1\, \mathcal{R}_n\, t_2 \quad \Longleftrightarrow \quad VarFrom(n,t_1) = VarFrom_n(n,t_2)$$

In the "initial graph" of Fig. 9, concerning node **n2** only two transitions are in relation: $\mathbf{A}(\mathbf{X})\, \mathcal{R}_{\mathbf{n2}}\, \mathbf{B}(\mathbf{X})$. Thus equivalence classes for node **n2** are
$\{\, \{\mathbf{A}(\mathbf{X}), \mathbf{B}(\mathbf{X})\},\ \{\mathbf{C}(\mathbf{X},\mathbf{Y})\}\, \}$

Algorithm The algorithm of Fig 10 uses the relation $\mathcal{R}_n$ to compute, for each node, (a) how many new nodes must be created (i.e. the number of equivalence classes), and (b) which transitions must be redirected to these new nodes.

This algorithm is based on a "**foreach**" loop which does not specify any order. However, $BoundVars$ sets must be computed in a precise order. This order is induced by dependencies in the definition of $VarFrom$ and the calculus of $BoundVars$. $BoundVars(n)$ depends on $VarFrom(n, \_)$. $VarFrom(n, \_)$ depends on $BoundVars$ sets of predecessors of node n. Therefore $BoundVars(n)$ depends on $BoundVars$ sets of the predecessors of node n. Since this algorithm does not consider "*backward*" transitions, the graph is acyclic, and then such an order exists.

Note that this algorithm does not use "*Backward*" transitions. Indeed, when going backward in the search process, the set of instantiated variable does not grow.

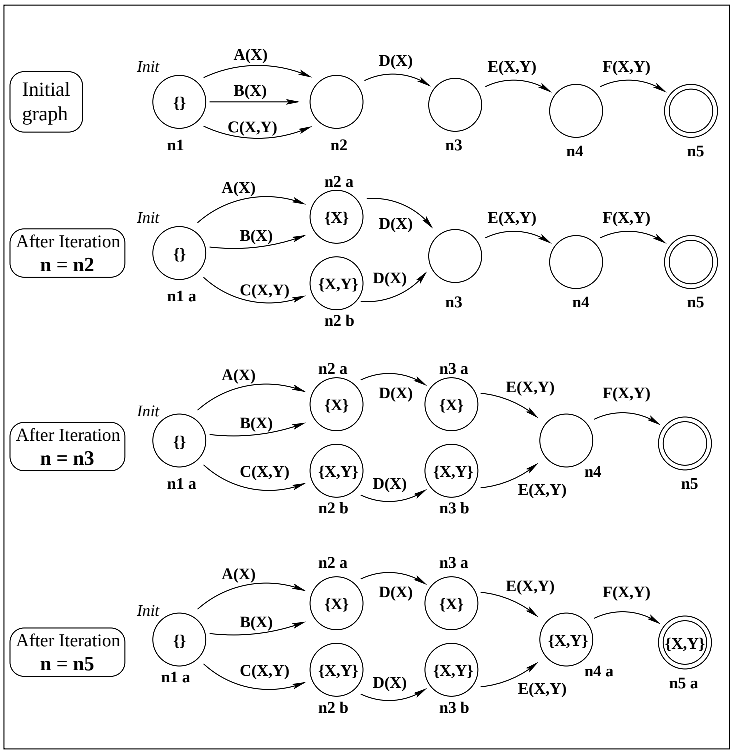

Figure9. Example of computation of *BoundVars* sets in a *SigGraph*

In Fig. 9, after iteration "n = **n2**", node **n2** is duplicated into two nodes because $BoundVars(n2)$ can be $\{X\}$ for transitions $A(X)$ or $B(X)$, or it can be $\{X, Y\}$ for transition $C(X, Y)$. For a similar reason, node **n3** is split. On the opposite, node $n4$ is not duplicated since both paths from *Init* to **n4** lead to the same set of bound variables. Node **n5** is not duplicated because there is only one incoming transition to this node.

4.5 Taking Variable Instantiation into Account in a *SigGraph*

BoundVars sets computed with the algorithm proposed in Sec. 4.4 are used in the last step of the construction of the *SigGraph*. These sets permit (a) to "freeze" evaluation of non-evaluable constraints due to a lack of variable instantiation, (b) to refine the placement of predicates in *SuchPred* sets in transitions, and (c) to find recurrent and non-recurrent nodes.

foreach node n **do**
  C = set of equivalence classes according to $\mathcal{R}_n$
  foreach $c_i \in C$ **do**
    Create a new node n_i where:
      Incoming transitions to n_i are those in c_i
      Outgoing transitions from n_i are those of n
      $BoundVars(n_i){:=}Varfrom(n,t)$ where t is a transition in c_i
  end_foreach
  Delete node n and all its incoming and outgoing transitions
end_foreach

Figure10. An algorithm to compute *BoundVars* sets in a *SigGraph*

"Freezing" non-evaluable constraints. Consider the following signatures where `f1` and `f2` are event fields, `c1` and `c2` are constants and `X` is a variable:

$S_1 =$ `[ f1 = c1, f2 != X ]` **then** `[ f1 = c2, f2 = X ]`
$S_2 =$ `[ f1 = c1, f2 != X ]` **and** `[ f1 = c2, f2 = X ]`

In signature S_1, when an event E occurs, such that `E(f1)=c1`, variable `X` has no value, and the constraint "`E(f2) != X`" cannot be evaluated. When looking for an instance of S_2, the same situation arises if an event E1, such that `E1(f1)=c1`, occurs before an event E2, such that `E2(f1)=c2`. These problems can be avoided by rewriting S_1 and S_2 as follows:

$S_1 =$ `[ f1 = c1, f2 = X ]` **then** `[ f1 = c2, f2 != X ]`
$S_2 =$ (`[ f1 = c1, f2 = X1 ]`
    and `[ f1 = c2, f2 = X2 ]`) **such_that** (`X_1 != X2`)

The initial expressions of both S_1 and S_2 are correct Sutekh expressions with a natural declarative semantics. Using results of $BoundVars$ set calculation, we automatically "freeze" constraints which are not evaluable, and delay their evaluation.

For each transition in the *SigGraph*, if a constraint (l, r, v) cannot be evaluated because v does not belong to $BoundVars$ of the origin of the transition:

- this constraint is transformed into "$l = v_{new}$" where v_{new} is a new variable.
- the variable v_{new} is added to $BoundVars$ sets of all nodes "posterior" to this transition.
- the predicate "$v_{new}\ r\ v$" is added in *SuchPred* sets of all transitions "posterior" to this transition .

This transformation of the *SigGraph* does not ensure that the new predicates added to *SuchPred* sets are evaluable where they are located. Refining the placement of *SuchPred* predicates is the purpose of the next item in this section.

Refining placement of *SuchPred* predicates. Figure 11 shows a signature which combines sequence, interleaving, negation and such_that statements. The

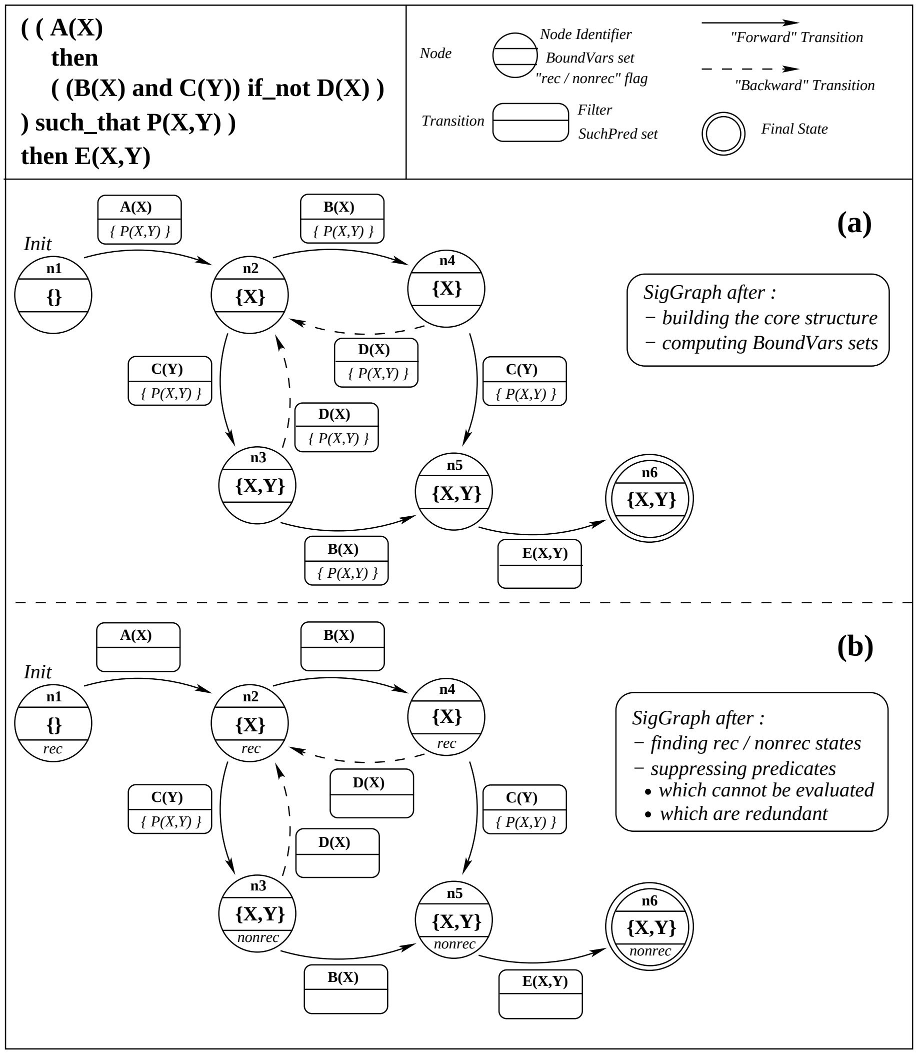

Figure11. Illustration of the last step of the *SigGraph* construction

first graph in the figure results from the first two steps of the construction of the *SigGraph* (building the core structure and computing $BoundVars$ sets). There are six nodes with their corresponding $BoundVars$ sets. Transitions are represented together with their filter and *SuchPred* set ($ProcTrig$ sets are omitted). Note that, during the construction of the core structure, the predicate $P(X, Y)$ has been attached to all transitions under the scope of the **such_that** statement.

Considering the first graph of Fig. (11a) two remarks can be done with respect to the predicates labeling the transitions: (a) $P(X, Y)$ cannot be evaluated on transition $(\mathbf{n1}, \mathbf{n2})$ because Y is not bound in **n1**, and (b) since predicate $P(X, Y)$ has been evaluated to true on transition $(\mathbf{n2}, \mathbf{n3})$, evaluating this predicate on transition $(\mathbf{n3}, \mathbf{n5})$ is redundant.

foreach transition $t = (n, n')$ **do**
  foreach predicate P in $SuchPred(t)$ **do**
    Let V be the set of variables used in P
    If ($V \not\subseteq BoundVars(n')$) **then** suppress P from $SuchPred(t)$
    If ($V \subseteq BoundVars(n)$) **then** suppress P from $SuchPred(t)$
  end_foreach
end_foreach

Figure12. An algorithm to delay non-evaluable predicates and remove redundant ones

The algorithm of Fig 12 removes both non-evaluable and redundant predicates. The test ($V \not\subseteq BoundVars(n')$) removes the predicate P if not enough information are available to evaluate P. In Fig. 11, this concerns $P(X, Y)$ on the transitions $(n1, n2)$, $(n2, n4)$, and $(n4, n2)$. The test ($V \subseteq BoundVars(n)$) removes the predicate P if there were enough information to evaluate P before performing this transition. In this case P has been evaluated before reaching the origin state of the transition. In Fig. 11, this concerns $P(X, Y)$ on the transitions $(n3, n2)$ and $(n3, n5)$.

Finding recurrent and non-recurrent states A state is marked "*nonrec*" if and only if it has at most one outgoing "$forward$" transition and its $BoundVars$ set is equal to the one of its successor through this transition. A state is marked "*rec*" in all others cases. Intuitively, this means that a state is said to be "$recurrent$" if, when transiting from this state, the IDS must be ready to detect another event which leads either to an other path in the graph, or to the same path with different variable instantiations.

This is shown in Fig. (11b). Node **n2** is recurrent: there exists two "$forward$" transitions ($B(X)$ and $C(Y)$) from this node. Nodes **n1** and **n4** are recurrent: transiting from both these nodes add variables in $BoundVars$ sets. Nodes **n3** and **n5** are non-recurrent: only one "$forward$" transition starts from each node, and transiting through it does not bind more variables. Node **n6** is non-recurrent since it has no successor.

5 Related Work

Sutekh provides a combination of functionalities at least as complete as the union of what is offered by the following declarative systems. Our chain of transformations from declarative signature specifications to operational rules gives an operational semantics to the declarative signatures independently of a given implementation.

Chakravarthy et al. define in [3] the semantics of Snoop, a *composite event* language in the context of active databases. The semantics of Sutekh extends the semantics of Snoop with logical variables to correlate events.

Lin et al. define MuSigs [11], a declarative signature language, as part of the ARMD project. In MuSigs, signatures are specified by a direct acyclic graph (DAG) where nodes are labeled with logical predicates describing constraints on events. The DAG structure allows temporal relations between events to be specified; variables in predicates express event correlation as in Sutekh. However, signatures in MuSigs cannot easily specify nested disjunctions, and we do not see any means to specify the equivalent of our **if_not**. Furthermore, the authors propose an iterative algorithm to look for instances of MuSigs signatures, but no details are given about the critical treatment of variable binding.

Gérard describes in [6] LaDAA, a declarative language where "simple conditions" (similar to Sutekh filters) are combined using sequence, disjunction and conjunction operators. Sutekh extends LaDAA by adding the **if_not** and the **such_that** constructs. The construction of the core structure of the *SigGraph* is inspired by the algorithm proposed in [6] to generate Russel rules for ASAX. Our algorithm, however, goes further with its static analysis of variable binding.

LAMBDA, presented by Cuppens and Ortalo [4], and ADeLe, proposed by Michel and Mé [13] are two languages providing frameworks to describe scenarios of attacks together with their conditions of execution, their impacts on the target system and their signatures. Fragments of these languages are dedicated to signature description. Sutekh can be seen as the union of these two fragments. Furthermore, we are not aware of an operational semantics of these two languages.

6 Conclusion

In this article we have described Sutekh, a declarative signature specification language, and have presented its semantics. We have also shown how to produce operational detection rules from declarative signatures. A number of technical problems, which must be meticulously considered when writing state-transition diagrams, have also been exhibited. In particular, partial order, together with negation and logical variables, can lead to instantiation problems. We have solved these problems by a static analysis of the evolution of variable binding during the detection. This analysis could also be done by hand in low-level specifications but it would be prone to mistakes. Our automated processes limits such risks and is well suited to further develop performance improvements.

Acknowledgment We thank M. Eluard and T. Jensen for their thorough proof-reading of earlier versions of this article and for fruitful discussions.

References

1. S. Axelsson. Intrusion Detection Systems: A Survey and Taxonomy. Technical Report 99-15, Depart. of Computer Engineering, Chalmers University, march 2000.
2. CERT Coordination Center. Advisory CA-1996-16 - Vulnerabilities in Solaris admintool. http://www.cert.org/advisories/CA-1996-16.html.
3. S.C. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite Events for Active Databases: Semantics, Contexts and Detection. In *Proc. of the 20th International Conference on Very Large DataBases (VLDB94)*, 1994.
4. F. Cuppens and R. Ortalo. LAMBDA: A Language to Model a Database for Detection of Attacks. In *Proc. of RAID'00*, LNCS vol. 1907, Springer, 2000.
5. S.T. Eckmann, G. Vigna, and R.A. Kemmerer. STATL: An Attack Language for State-based Intrusion Detection. In *Proc. of ACM Workshop on Intrusion Detection, Athens, Greece*, Nov 2000.
6. F. Gérard. Définition et implémentation d'un langage d'analyse d'audit trails. Faculté Universitaire Notre-Dame de la Paix, Namur, Belgium - http://www.info.fundp.ac.be/~amo/publications.html, 1998.
7. N. Habra, B. Le Charlier, A. Mounji, and I. Mathieu. ASAX: Software Architecture and Rule-based Language for Universal Audit Trail Analysis. In *Proc. of (ESORICS)'92*. Springer-Verlag, 1992.
8. K. Ilgun and A. Kemmerer ans P.A. Porras. State Transition Analysis: A Rule-Based Intrusion Detection Approach. *IEEE Transactions on Software Engineering*, 21(3):181–199, march 1995.
9. S. Kumar. *Classification and Detection of Computer Intrusions*. PhD thesis, Purdue University, West Lafayette, 1995.
10. S. Kumar and E.H. Spafford. A Pattern Matching Model for Misuse Intrusion Detection. In *Proc. of the 17th National Computer Security Conference*, 1994.
11. J.-L. Lin and X. Sean Wang amd S. Jajodia. Abstraction-Based Misuse Detection: High-Level Specifications and Adaptable Strategies. In *Proc. of the 11th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1998.
12. U. Lindqvist and P.A. Porras. Detecting Computer and Network Misuse Through the Production-Based Expert System Toolset (P-BEST). In *Proc. of the IEEE Symposium on Security and Privacy*, 1999.
13. C. Michel and L. Mé. ADeLe: an Attack Description Language for Knowledge-based Intrusion Detection. In *Proc. of the 16th International Conference on Information Security*, 2001.
14. A. Mounji. *Languages and Tools for Rule-Based Distributed Intrusion Detection*. PhD thesis, Facultés Notre-Dame de la Paix - Namur (Belgium), 1997.
15. P.G. Neumann and P.A. Porras. Experience with Emerald to Date. In *Proc. of the 1st USENIX Workshop on Intrusion Detection and Network Monitoring*, 1999.
16. Sun MicroSystem. SunSHIELD Basic Security Module Guide, Feb 2000. Part Number 806-1789-10.