Advanced **S**ecurity **A**udit Trail Analysis on uni**X**
(ASAX also called SAT-X)
Implementation design of the NADF Evaluator

Naji Habra Baudouin Le Charlier
Abdelaziz Mounji,
Institut D'Informatique,
FUNDP,
rue Grangagnage 21,
5000 Namur
E-mail: {nha, ble, amo}@info.fundp.ac.be

September 26, 1994

Contents

Chapter 1

Introduction

This report is devoted to the implementation design of the NADF evaluator. Knowledge of [1] is assumed.

The report consists of three main parts. Chapter 2 describes general data structures used by the evaluator. Chapter 3 explains how the rule based language is implemented by means of an efficiently implementable abstract machine language. Chapter 4 presents the syntax analyser able to translate the rule based language into abstract machine language form. Annex A defines system parameters and how to maintain them, Annex B gives the list of compiler error messages their description and how to correct them.

Chapter 2

Implementation design of the current environment

2.1 The current record

During the analysis process, the audit trail is likely to be heavily accessed by the active rules. This is due to the fact that most of the time the system is evaluating expressions involving the current record fields. As a result, access to the current record must be designed and organized with care. The overall system performance depends strongly on efficient access to the current record.

2.1.1 Current record table

Before being processed for all current rules, the current record is pre-processed to update an indirection table that will speed up access to it. This processing is similar to what is currently done in the SATUT and is detailed below:
The current record is accessed from a location called the *current record buffer*. An entry of the indirection table is composed of two fields: an audit data id and a pointer inside the current record buffer.
At compile time, all declared rules are scanned for references to the current record fields. A list or "catalogue" of all these references is made. Every subsequent reference to the current record will necessarily involve one of this catalogue items. The corresponding field identifiers are stored in this indirection table with ascending order. At this step, each first component of each table entry contains a field identifier (there may be some unused area at the end of the table.) All indirection table entries are located at a fixed storage location well known to the system at compile time.
During the audit trail analysis, when an audit record becomes current, it is scanned in parallel with the indirection table so as to update the audit data pointers. Note that this scanning takes place only once and that during all the processing of the current record, the indirection table remains unchanged. Finally every reference to an audit data in the rule internal codes will be represented by the location of the corresponding indirection table

entry. This way, audit data are directly available[1].
Given an audit data identifier in the indirection table, if the current record contains the corresponding audit data field, then the pointer part of the indirection table entry is assigned a pointer to this audit data; otherwise, it is assigned a special value noted here **non-present** to indicate such a case. In fact this special value is the address of the null string (' ') which is compatible with the language semantics. More details about the current record table design as well as the functions for initializing and updating it can be found in 4.3.2 where the C declarations are also provided.

2.1.2 Audit data representation

Integers are simply represented by their respective values. Strings of bytes are represented by pointers to areas having the following form:

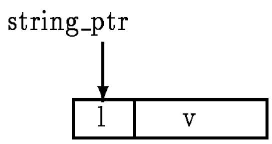

Figure 2.1: Audit data representation

where l is the string length and v is the string value.
This string representation is adequate for managing their variable length and is more general than null-ended string representation.

Notation

Throughout this document, *ri(object)* denotes the internal representation of the object *object*

Example

If s is a string of bytes, then *ri(s)* represents a pointer to a dynamic variable containing the length and value of s.

Remark

A null string ' ' is represented by:

[1] In fact, the rule internal codes are modified in parallel with the indirection table using a delayed pointer assignment technique which will be detailed in the next chapter.

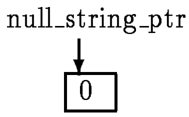

Figure 2.2: Null string

2.1.3 Representation of a list of values

Let L = [v_1,...,v_m] be a list of values. These are either string of bytes or integers. Let *ri(v_1)*, ..., *ri(v_m)* be their respective internal representations. Therefore, the internal representation of the list L is a contiguous storage area built up with the internal representations of v_1,...,v_m:

Figure 2.3: Liste of values representation

Example

Let v_1 = 123, v_2 = 'TOTO', v_3 = 127; then *ri([v_1, v_2, v_3])* has the form:

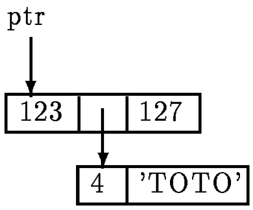

Figure 2.4: Example of a list of values representation

2.2 Representation of the local environment

The local environment is basically a set of variables. These variables may be either actual parameters or local variables. Consider a rule having x_1, ..., x_m as actual parameters and y_1, ..., y_n as local variables. The internal representation of the corresponding local environment are the two storage areas LVA (*Local Variables Area*) and APA (*Actual Parameters Area*):

APA:	ri(v_1)	...	ri(v_m)

LVA:	ri(u_1)	...	ri(u_n)

Figure 2.5: Representation of the local environment

where v_i and u_j denote the current values of x_i and y_j (resp.) $(1 \leq i \leq m$ and $1 \leq j \leq n)$.

Note

The two storage areas, APA and LVA, are not necessarily contiguous.

2.3 The three sets of rules

Recall that at any time during the analysis process, there exist three subsets of rules:

- rules triggered off for the current record;
- rules triggered off for the next record;
- rules triggered off for completion of the analysis.

All of these sets have the same operational properties so they are treated the same way.
The primitive operations defined for these sets are:

- creating an empty set of rules;
- adding a rule to a set of rules;
- removing a rule from a set of rules.

Since the execution of the evaluator is not affected by the order in which rules are added or removed from a set of active rules, the latter can be indifferently modeled as a stack or a queue. For the simplicity of the implementation, a set of rules is modeled as a stack and implemented by a linked list. A particular *cell* of this list is a dynamic area related to a particular rule (instance of a rule schema).

At the logical level, a cell contains:

- a pointer towards the internal code of the rule;

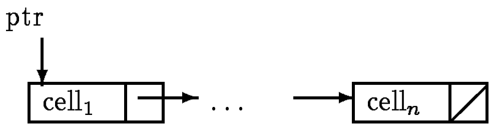

Figure 2.6: A set of active rules

- a pointer towards an area containing the values of the actual parameters. (this area is copied into the APA when the processing of the rule is started);
- a pointer to the next cell in the list.

The actual representation of such a linked list is detailed in 3.4.7.

2.4 Access to actual parameters and local variables

During the execution of a particular rule, local variables and actual parameters are frequently accessed. Therefore, actual parameters and variables must be located as quickly as possible. Considering the implementation of DStrig, an actual parameter is accessed in the following steps:

determine the base address of the related rule;

determine the relative (or logical) address of the actual parameter;

the location is the sum of the two.

This is a quite acceptable implementation and relative to machine address conversion can be done fairly quickly. However, we can still improve on the access speed by having the list of actual parameters and local variables be located at the particular storage areas Actual Parameter Area and Local variable Area with well known fixed addresses[2]. Consequently, the machine (or absolute) address can be determined at compile time and once for all. The resulting access speed is then optimal.
When a new rule is to be executed against the current record, the area containing its actual parameters is first copied to the APA before been used. But this is fairly a little overhead.

[2]This access speed improvement can be achieved without sacrificing too much machine independence because these fixed addresses can be made localized so as they can be easily tuned to a particular machine configuration.

Chapter 3

Implementation design of the language constructs

3.1 Introduction

Before discussing how the language constructs (i.e. rules, actions, expressions) will be implemented, it is useful to introduce an interesting notation for representing these language constructs. Throughout the remainder of this document, we will make use of these notations in order to provide a clear and machine independent way to represent how the language constructs are coded. This notation makes use of basic constructs which constitutes the (machine) language of an abstract machine.

3.2 Box notation and abstract machine language

A *box notation* represents either an expression or an action. There exist two kinds of boxes: *elaborate* boxes and *terminal* boxes. Elaborate boxes represent compound constructs so they can be refined by breaking them into several boxes whose execution amounts to executing the original ones. Terminal boxes cannot be further refined. A terminal box represents a (basic) instruction of the abstract machine. It is internally coded as a set of contiguous storage locations. For each type of constructs, we will give the corresponding box notation.
A terminal box contains the following components:

1. an identifier that indicates an action to be performed (e.g. , assign, multiply, add, jump, ... etc);

2. a list of operands that depends on the action. These operands are absolute object addresses;

3. a list of *branch addresses* that indicates which abstract instruction will be executed next. In fact there is always only one branch address except for boolean expressions where there are two possible constructs

to be executed next depending on the truth value of the expression.

As a result, a complex construct can be progressively refined in order to obtain a chained list of terminal boxes that forms a program in the abstract machine language.

Notation

Let *cons* be a construct (either a rule, an action or an expression). Suppose that the next construct to be executed afterwards is at address α and that β is the address of the first abstract machine instruction to be executed for *cons*, then the box notation for *cons* is:

$$\beta : \quad \boxed{\mathrm{ri}(\mathrm{cons},\alpha)}$$

Figure 3.1: Abstract machine statement

where the box stands for the set of all abstract machine instructions constituting the internal code for *cons*.

3.3 Internal codes

This section deals with internal codes for expressions. For each kind of expression, we provide the corresponding internal code as well as the graphical notation. A formal proof of the consistency with the abstract semantics is presented as an example.

3.3.1 Simple expressions

Recall that

expr ::= <literal> | <variable> | <parameter> | <field_name>.

In this case, an internal representation already exists and need not be created. For instance, a local variable is allocated a storage memory inside the LVA area and an audit data is referenced through a pointer inside the current record table. Consequently, the value of a literal, a variable, a parameter or a field name is already available and no internal code is generated for its evaluation. Let α be the address of the next statement to be executed and β the internal representation address of the literal, variable, parameter or field name. Since the evaluation amounts to do nothing, we have $\alpha = \beta$. It follows from the above discussion a notation convention given hereafter: If $\alpha = \beta$, no memory storage is allocated for the statement internal representation in question. However, for the sake of generality, we still use the notation in Figure 3.2 (then the box stands for an *empty* set of abstract instructions)

$$\beta \ \boxed{\mathrm{ri}(\mathrm{cons},\alpha)}$$

Figure 3.2: Convention

3.3.2 Compound arithmetic expressions

First of all, let us introduce the parameters used by such a statement:

1. *expr* is the expression to be evaluated with respect to a certain environment. It is supposed to return integer or string value;

2. α is the location of the next construct to be executed;

3. the evaluation of an expression often involves the evaluation of subexpressions. Those subexpression values must be saved somewhere in order to be retrieved and used later. As a result, we provide a special storage area called the *working area WA* where these temporary results are stored.

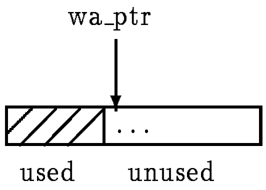

Figure 3.3: The working area

δ_{in} denotes a pointer to the first free (unused) element of WA *before* execution of *expr*;

4. β is the location of the internal code corresponding to *expr* itself;

5. γ is the location of the value returned by the expression;

6. δ_{out} denotes the pointer to the first free element of WA *after* execution *expr*.

Under the above conditions, the graphical notation chosen for *expr* is:

$$\beta : \quad \boxed{\mathrm{ri}(\mathrm{expr},\alpha,\delta_{in})}$$

Figure 3.4: Box notation for integer and string expressions

The resulting γ and δ_{out} are functions of the structure (simple or compound) and the type (string or integer) of the expression.

1. suppose that *expr* is either a literal, a parameter, a variable or a field name. In this case, an internal code for *expr* already exists so there is no need to generate it. Also, no intermediary results are needed for evaluating the expression because its value is directly available (inside the current record, the local environment or somewhere else.) As a result:

   $$\alpha = \beta,\ \delta_{out} = \delta_{in} \text{ and } \gamma \text{ is the location of } ri(expr);$$

2. otherwise, the expression is a compound one and intermediary results have to be stored in WA before evaluating the rest of the expression. In this case,

   $$\delta_{out} = \delta_{in} + lv;$$

   where *lv* denotes a value length, i.e., the pointer to the first free element must be advanced by one position.

Let us now examine in detail the different possible cases. Later, we will see how these cases can be reduced to a single one:

Suppose that

$$expr ::= expr_1\ \omega\ expr_2 \qquad \text{where } \omega \text{ is an arithmetic operator.}$$

Let $\delta_{in,i}$ and $\delta_{out,i}$ be the pointers to the next free element of WA respectively before and after evaluation of expr_i (i=1, 2).

1. $\delta_{out,1} > \delta_{in,1}$ $(= \delta_{in})$ and $\delta_{out,2} = \delta_{in,2}$

   In this case, $expr_1$ corresponds to case 2 and $expr_2$ corresponds to case 1 above. The resulting decomposition is :

   $$\begin{array}{lll} & \beta_1\text{:} & \beta_2\text{:} \\ \beta\text{:} & \boxed{\text{ri}(\text{expr}_1, \beta_2, \delta_{in})} & \Big\| \begin{array}{|c|c|c|c|c|}\hline \omega & \delta_{in} & \delta_{in} & \gamma_2 & \alpha \\ \hline\end{array} \Big\| \end{array}$$

   where $\gamma = \delta_{in}$, $\delta_{out} = \delta_{in} + vl$ (a value length).

   The second box means applying the operator ω to the values located in δ_{in} and γ_2 and storing the result in address δ_{in}; the next construct to be executed being at location α;

$$\beta: \overset{\beta_1:}{\boxed{\mathrm{ri}(\mathrm{expr}_2,\beta_2,\delta_{in})}} \quad \Big\| \overset{\beta_2:}{\begin{array}{|c|c|c|c|c|}\hline \omega & \delta_{in} & \gamma_1 & \delta_{in} & \alpha \\ \hline\end{array}} \Big\|$$

2. $\delta_{out,1} = \delta_{in}$ and $\delta_{out,2} > \delta_{in}$

   This case is symmetrical to the first one where $expr_1$ and $expr_2$ play opposite roles. The decomposition is then:

   $\gamma = \delta_{in}$, $\delta_{out} = \delta_{in} + vl$;

3. $\delta_{out,1} = \delta_{out,2} = \delta_{in}$
   Both expressions correspond to the above first case. Let γ_1 and γ_2 be the addresses of their respective values. The resulting decomposition is then:

   $$\beta: \Big\| \overset{\beta_1:}{\begin{array}{|c|c|c|c|c|}\hline \omega & \delta_{in} & \gamma_1 & \gamma_2 & \alpha \\ \hline\end{array}} \Big\|$$

   $\gamma = \delta_{in}$, $\delta_{out} = \delta_{in} + vl$;

4. $\delta_{out,2} > \delta_{out,1} > \delta_{in}$
   In this last case, both expressions correspond to the second case. The value of $expr_1$ and $expr_2$ are at δ_{in} , δ_{out} respectively. The decomposition is: $\gamma = \delta_{in}$, $\delta_{out} = \delta_{in} + vl$.

   $$\overset{\beta_1:}{\boxed{\mathrm{ri}(\mathrm{expr}_1,\beta_2,\delta_{in})}} \quad \overset{\beta_2:}{\boxed{\mathrm{ri}(\mathrm{expr}_2,\beta_3,\delta_{out})}} \quad \Big\| \overset{\beta_3:}{\begin{array}{|c|c|c|c|c|}\hline \omega & \delta_{in} & \delta_{in} & \delta_{out} & \alpha \\ \hline\end{array}} \Big\|$$

As announced before, it is in fact possible to have all of these four cases be reduced to a single one.
Let γ_1 and γ_2 be respectively the location of the values of $expr_1$ and $expr_2$. The decomposition is shown in Figure 3.5. Again, $\gamma = \delta_{in}$, $\delta_{out} = \delta_{in} + vl$;

3.3.3 Compound conditions

Let *cond* be a compound condition of the form:

$$cond_1 \ \omega \ cond_2 \qquad \text{where } \omega \in \{\textbf{or}, \textbf{and}\}$$

Let α_t (resp. α_f) be the address of the next statement to be executed if *cond* is evaluated to **true** (resp. **false**).

$$\beta\!: \underset{}{\overset{\beta_1:}{\boxed{\mathrm{ri}(\mathrm{expr}_1,\beta_2,\delta_{in})}}} \quad \overset{\beta_2:}{\boxed{\mathrm{ri}(\mathrm{expr}_2,\beta_3,\delta_{out})}} \quad \overset{\beta_3:}{\Big\|\,\boxed{\omega}\boxed{\delta_{in}}\boxed{\gamma_1}\boxed{\gamma_2}\boxed{\alpha}\,\Big\|}$$

Figure 3.5: General case

$$\beta\!: \overset{\beta_1:}{\boxed{\mathrm{ri}(\mathrm{cond}_1,\beta_2,\alpha_f)}} \quad \overset{\beta_2:}{\boxed{\mathrm{ri}(\mathrm{cond}_2,\alpha_t,\alpha_f)}}$$

Figure 3.6: Conjunction

Suppose $\omega =$ **and**, the corresponding graphical notation is shown in Figure 3.6. In case $\omega =$ **or**, we have the representation given in the figure 3.7. Note that evaluation is terminated as soon as the truth value is known without taking into account the rest of the compound condition. For instance, if ω is **and** and $cond_1$ evaluates to **false**, then *cond* is evaluated to **false** whatever the value of $cond_1$ is and no matter if this evaluation is defined or not.

3.3.4 Consistency with the abstract semantics

The above implementation design must still be proved coherent with the abstract semantics given in [1]. However, although this can be very interesting, the purpose of this document is not to provide a formal proof of the consistency of each construct internal representation with its abstract semantics. Nevertheless, a sample proof schema for conditional expressions is given below in order to illustrate the main ideas. This proof is straightforward thanks to the box notation.

Base case: Simple condition

The base case is trivial since the value of a literal (X_literal or C_literal), a variable, parameter or a field name (audit data) coincides with the value of its internal representation.

Inductive case: Compound conditions

Let *cond* be a condition of the form:

$$\beta\!: \overset{\beta_1:}{\boxed{\mathrm{ri}(\mathrm{cond}_1,\alpha_t,\beta_2)}} \quad \overset{\beta_2:}{\boxed{\mathrm{ri}(\mathrm{cond}_2,\alpha_t,\alpha_f)}}$$

Figure 3.7: Disjunction

$$cond_1 \textbf{ and } cond_2$$

This is represented by the box notation in Figure 3.8

$$\beta\text{:}\ \overset{\beta_1\text{:}}{\boxed{\mathrm{ri}(\mathrm{cond}_1,\beta_2,\alpha_f)}}\quad \overset{\beta_2\text{:}}{\boxed{\mathrm{ri}(\mathrm{cond}_2,\alpha_t,\alpha_f)}}$$

Figure 3.8: Conjunction

We have to show that the execution of the abstract statement yields the same truth value as the one given by the abstract semantics of compound conditions (see [1]). That is to say, when the value is **true** (resp. **false**) the branch address must be α_t (resp. α_f). By the induction hypothesis, since $cond_1$ evaluates to **true** the branch address is β_2. Now, according to the box notation, the branch address of the whole expression is that of Figure 3.9.

$$\overset{\beta_2\text{:}}{\boxed{\mathrm{ri}(\mathrm{cond}_2,\alpha_t,\alpha_f)}}$$

Figure 3.9: Induction case

Again, by the induction hypothesis and since $cond_2$ evaluates to true this branch address is α_t.

Similar proof can be given when *cond* is evaluated to **false**.

3.3.5 Relational expressions

Let

$$cond ::= expr_1\ \omega\ expr_2$$

be a relational expression where ω denotes a relational operator. The two arithmetic expressions must be evaluated; this has been the purpose of a previous section, so we don't have to bother how this is actually done. However, the relevant parameter to be considered here is which construct is to be executed next depending on the result of the comparison. Let α_t and α_f be respectively the address of the next construct when *cond* is evaluated to **true** and when it is evaluated to **false**. If δ_0 is the value of *waptr* (pointer to the next free element in the work area) before the evaluation and γ_1 and γ_2 the locations of the values returned by $expr_1$ and $expr_2$, then, the box notation of *cond* is that of Figure 3.10. The semantics of the last box (which is a terminal one) is to compare the arguments located at γ_1 and γ_2. If the value of this comparison is **true**, the construct at address α_t is the one to be executed next; otherwise, the one at address α_f is executed.

$$\beta\colon\ \overset{\beta_1:}{\boxed{\mathrm{ri}(\mathrm{expr}_1,\beta_2,\delta_0)}}\quad \overset{\beta_2:}{\boxed{\mathrm{ri}(\mathrm{expr}_2,\beta_3,\delta_{out,1})}}\ \Big\|\overset{\beta_3:}{\boxed{\omega}\boxed{\gamma_1}\boxed{\gamma_2}\boxed{\alpha_t}\boxed{\alpha_f}}\Big\|$$

Figure 3.10: Box notation for relational expressions

Note

It is interesting to note that use of arguments in the box notation allows simple and constructive reasoning about the representation.

3.3.6 Simple expression involving the current record

Such an expression has the form:

$$\textit{cond} ::= \textbf{present}\ \textit{field\_name}$$

This condition is represented by the following terminal box whose arguments are:

- **present** is the box type indicator;
- α_t and α_f are the branch addresses;
- γ is merely the location of the current record table entry corresponding to this field name. Access to the current record is detailed in 4.3.2.

$$\beta\colon \Big\|\boxed{\text{present}}\boxed{\gamma\quad}\boxed{\alpha_t}\boxed{\alpha_f}\Big\|$$

Figure 3.11: Box notation for **present** *field_name*

3.4 Actions

3.4.1 Introduction

Internal representation of actions and expressions are very similar. The box notation and naming conventions apply as well. A global view of how actions are executed against the current record and the current environment will be given in the next chapter.

3.4.2 Skip statement

No storage is allocated for the empty statement internal code and $\beta = \alpha$ so Figure 3.4.2 is equivalent to $\beta = \alpha$.

$$\beta\text{: } \boxed{\text{ri(skip, } \alpha)}$$

3.4.3 Assignment

Assignment internal representation is equally simple. Let

$$x ::= expr$$

be the assignment to be represented, γ the address of the value returned by the expression *expr*, *adr(x)* the address of the variable x, the box notation is given in Figure 3.12

$$\beta\text{: } \overset{\beta_1\text{:}}{\boxed{\text{ri(expr,}\beta_2)}} \quad \overset{\beta_2\text{:}}{\begin{array}{|c|c|c|c|}\hline \text{assign} & \text{adr(x)} & \gamma & \alpha \\ \hline\end{array}}$$

Figure 3.12: Box notation for assignment statement

The second box (which is a terminal one) means assigning the content of address γ to the content of address *adr(x)*. We distinguish two types of assignment: assignment of integer values and assignment of addresses so the operator *assign* have two values: *assign_int* and *assign_addr*.

3.4.4 Conditional and repetitive actions

The two types of actions are treated here simultaneously because the corresponding internal representations are almost identical thanks to the box notation.
Let $b_1 \rightarrow$ a$_1$; ...; b$_n \rightarrow a_n$ be the set of guarded actions contained in either repetitive or conditional actions. The box notation corresponding to the conditional action:

$$\textbf{if } b_1 \rightarrow a_1; \ldots; b_n \rightarrow a_n \textbf{ fi}$$

is given in Figure 3.13:

$$\begin{array}{lccc} \beta\text{:} & \overset{\beta_1\text{:}}{\boxed{\text{ri(b}_1,\alpha_1,\beta_2)}} & \ldots & \overset{\beta_n\text{:}}{\boxed{\text{ri(b}_n,\alpha_n,\alpha)}} \\ \alpha_1\text{:} & \boxed{\text{ri(a}_1,\alpha)} & \ldots & \overset{\alpha_n\text{:}}{\boxed{\text{ri(a}_n,\alpha)}} \end{array}$$

Figure 3.13: Box notation for conditional actions

while the corresponding repetitive action

$$\textbf{do } b_1 \rightarrow a_1; \ldots; b_n \rightarrow a_n \textbf{ od}$$

has the box notation shown in Figure 3.14

$$\begin{array}{llcl}
 & \beta_1\text{:} & & \beta_n\text{:} \\
\beta\text{:} & \boxed{\mathrm{ri}(\mathrm{b}_1,\alpha_1,\beta_2)} & \ldots & \boxed{\mathrm{ri}(\mathrm{b}_n,\alpha_n,\alpha)} \\
 & & & \alpha_n\text{:} \\
\alpha_1\text{:} & \boxed{\mathrm{ri}(\mathrm{a}_1,\beta)} & \ldots & \boxed{\mathrm{ri}(\mathrm{a}_n,\beta)}
\end{array}$$

Figure 3.14: Box notation for repetitive actions

The power of the box notation shows how semantically different actions can be represented almost the same way. As a result, if either internal representation is available, it costs almost nothing to implement the other one.

3.4.5 Compound action

Assuming that we know how to represent a set of actions $\{\mathrm{a}_1, \ldots, \mathrm{a}_n\}$, it is easy to represent the corresponding compound action i.e.:

$$\textbf{begin } \mathrm{a}_1; \ldots; \mathrm{a}_n \textbf{ end}$$

Let β_i denote the location of the action a_i internal code, the box notation of the compound action is the following:

$$\begin{array}{lllcl}
 & \beta_1\text{:} & \beta_2\text{:} & & \beta_n\text{:} \\
\beta\text{:} & \boxed{\mathrm{ri}(\mathrm{a}_1,\, \beta_2)} & \boxed{\mathrm{ri}(\mathrm{a}_2,\, \beta_3)} & \ldots & \boxed{\mathrm{ri}(\mathrm{a}_n,\, \alpha)}
\end{array}$$

Figure 3.15: Box notation for compound action

where α denote the location of the next statement to be executed after the whole compound action is executed.

3.4.6 Representation of a rule

A rule has the form:

rule ::= **rule** *rule_name*(*parameter_list*);
local_var;
a

Its internal code ri(*rule_name*) is in Figure 3.16 where *nil* is a special (ending) address which means that the whole action part has been executed. The control has then to be returned to the next rule to be executed (i.e., the rule at the beginning of either DStrig or DScompl).

β: $\boxed{\text{ri(a, nil)}}$

Figure 3.16: Box notation of a rule

3.4.7 Rule triggering

Rule triggering has in part been discussed in a previous section; more details about how a rule triggering is actually represented is the purpose of the present section.
When dealing with a rule triggering, the main objects to be considered are:

- the action part of the corresponding rule schema;
- the list of actual parameter values.

These components define what is called a *rule triggering descriptor*. In fact the triggering of a rule amounts to build and add such a descriptor to one of the linked list corresponding to *DStrig, DSnext* or *DScompl*. The action part is represented by its box notation as seen before. The list of actual parameter values is represented by a storage area containing the representation of this list (see 2.3). If an actual parameter is a field name[1] the corresponding string in the current record must be saved in the rule triggering descriptor. Otherwise, when the next audit record is read, this string will be lost. The detailed structure of a rule triggering descriptor is given in Figure 3.17 where

s

lg	paramValues	savedStrings	boxPtr	nextCell

Figure 3.17: format of a rule triggering descriptor

nextCell is a pointer towards the next rule descriptor, *boxPtr* is a pointer to the rule internal code, *paramValues* is the list of actual parameter values, *savedStrings* is a contiguous area of saved string internal representations and finally, *lg* is the length in bytes of the descriptor minus the length of *nextCell*.
Example
Consider the following rule triggering:

trigger off for_next r(i, 'toto', EVENT);

Suppose that $i = 123$ and the value of EVENT (a field name) just *before* execution of the above rule triggering is (4, 'open'), then the corresponding rule triggering descriptor is that of Figure 3.18.

[1] In fact, if a string variable was previously assigned a the value of a field name, this string must also be saved. Abstract interpretation techniques can be used to detect such cases but for the prototype evaluator, we choose to save all string variables as well as field names.

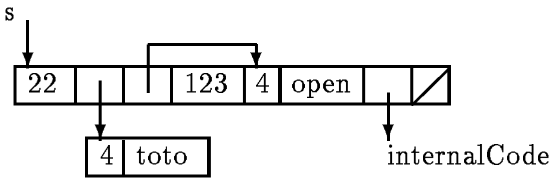

Figure 3.18: a sample rule triggering descriptor

The internal code generated for a rule triggering is a sequence of boxes that build up such a descriptor and push it in the appropriate stack of rule triggering descriptors (DStrig, DSnext or DScompl).
Given the rule triggering:

$$\textbf{trigger off } \textit{trigger\_mode rule\_name}(\textit{expr}_1, \ldots, \textit{expr}_n)$$

the internal code generated for it is shown in Figure 3.19. The effect of the

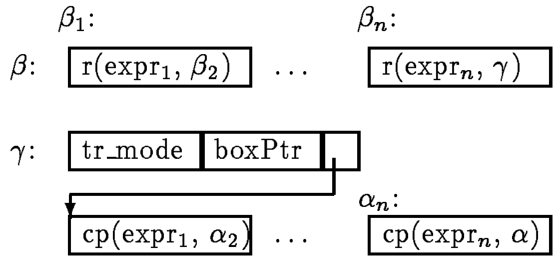

Figure 3.19: Box notation for rule triggering

sequence of boxes of the first line is to evaluate in sequence the expressions $\textit{expr}_1, \ldots, \textit{expr}_n$ and to store their values in the Working Area. The box of the second line allocates a memory area (whose size was computed by the boxes in the first line) for the rule triggering descriptor to be built; this descriptor is then pushed in the appropriate stack. Finally, the sequence of boxes in the third line are responsible for:

- copying the values of $\textit{expr}_1, \ldots, \textit{expr}_n$ from the Working Area to the descriptor;
- copying all strings that must be saved in the descriptor.

The content of a box in the first or the third line depends on the type of $\textit{expr}_i$ and whether it must be saved in the descriptor or not.

- if $\textit{expr}_i$ is of type **integer**
  - if the evaluation of $\textit{expr}_i$ requires the generation of at least one box, *r(*$\textit{expr}_i$*,* β_{i+1}*)* is simply *ri(*$\textit{expr}_i$*,* β_{i+1}*)*;

– otherwise, *expr*$_i$ is a constant or a variable name and no box is generated for its evaluation, in this case, the value of *expr*$_i$ must be moved to the W.A. so that *r(expr*$_i$*,* β_{i+1}*)* is the box of Figure 3.20.

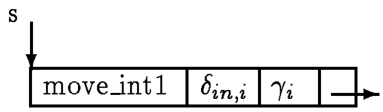

Figure 3.20: move box

in both cases, *cp(expr*$_i$*,* α_{i+1}*)* is the box of Figure 3.21 which copies the value of *expr*$_i$ from the W.A. to the descriptor

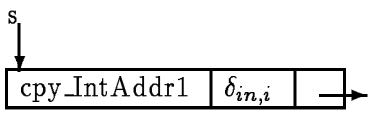

Figure 3.21: a box to copy an integer

- otherwise *expr*$_i$ is of type **string**

  – if *expr*$_i$ is a constant (C or X-literal) no box is needed for its evaluation since its value is in the table of constants and for the same reason, its value need not be saved in the descriptor. As a result, *r(expr*$_i$*,* β_{i+1}*)*is the box of Figure 3.22 which moves the

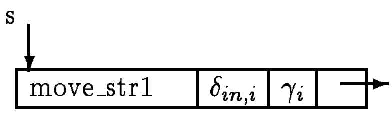

Figure 3.22: a box to copy a string address

  value of *expr*$_i$ from the constant table to the W.A. and *cp(expr*$_i$*,* α_{i+1}*)* is the box of Figure 3.23 that copies the address of the string in the descriptor;

  – otherwise, the string must be moved to W.A. and then saved in the descriptor. As a result, *r(expr*$_i$*,* β_{i+1}*)* is the two boxes of Figure 3.24 where the effect of the second box is to increment the accumulated size of the area to be allocated for the rule descriptor while *cp(expr*$_i$*,* α_{i+1}*)* is the box of Figure 3.25 which copies the the string *itself* in the descriptor and then copies the address of this string in the descriptor.

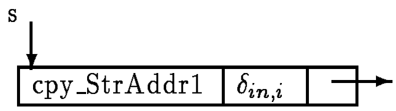

Figure 3.23: copy a string address in the descriptor

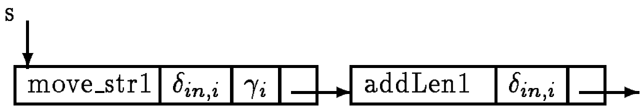

Figure 3.24: evaluate a string and compute its length

A single rule may obviously be triggered off more than once for the current record (but with perhaps a different actual parameter values). In this case, the action part would be duplicated over the linked list (either *DStrig, DSnext* or *DScompl*). In order to optimize the storage, the chosen implementation is to allocate a single storage area for the action part. Therefore, each rule triggering descriptor contains a pointer to this area. However, the list of actual parameter values will be stored in this descriptor.
A detailed description of the semantic of each of the above boxes is given in section 3.5.

3.4.8 Predefined procedures

The rule-based language provides explicitly two types of arguments passing:

- "call by value"
- "call by reference"

which have the same common meaning as for other programming languages like Pascal. So, when called, a predefined procedure may be handed a value of its arguments in temporary variables or their addresses. A parameter passed by value may be any compound expression while it must be a variable when passed by reference.
More precisely, a value parameter may have either an integer type or be a pointer to a string of bytes, while a passed by reference parameter may be either a pointer to an integer or the address of a pointer to a string. The description of the argument passing type is part of a predefined procedure

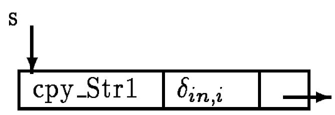

Figure 3.25: evaluate a string and compute its length

specification. As a result, in addition to the predefined procedure code, contained in the library, a description of each of its argument passing type is stored so that calls to this procedure are performed in accordance with this description.
In fact the predefined library supports two kinds of predefined routines: those with fixed-length argument list and those with variable-length argument list. For the latter, that means the number and types of these arguments may vary. In this case, the type of argument passing (value or reference) is the same for all arguments.
The predefined library is implemented by two tables. The first one *tfp_descr[]* is an array of predefined routine descriptors and the second *targ_descr[]* is an array of argument descriptors.
A predefined routines descriptor is a structure of the following fields:

- routine name;
- routine address;
- routine arity;
- the type of the returned value;
- index in *targ_descr[]* of the first argument descriptor related to this routine.

If a routine has a variable-length argument list, its arity is (-1) by convention and the last field of its descriptor (index in *targ_descr[]*) must be interpreted as the passing mode which is common to all its arguments.
An argument descriptor (i.e., an entry in *targ_descr[]*) is a structure of the following fields:

- the passing mode (reference or value);
- argument type.

Here is the C declarations of these two tables:

```
typedef struct {             /* predefined routine descriptor*/
      char        *name;     /* function or procedure name   */
      retfp       fp_ptr;    /* its address                  */
      int         narg;      /* =-1 when variable arity      */
      enum type   rettype;   /* type of the returned value   */
      int         argp;      /* index in the table targ_decsr[] */
} fp_descr;
fp_descr tfp_descr[tfp_descr_lg];
enum passtype {value, ref};  /* the passing modes            */

typedef struct {             /* an argument descriptor       */
      enum passtype  tpass;  /* the passing type             */
      enum type      targ;   /* the type                     */
} arg_descr;
arg_descr targ_descr[targ_descr_lg];
```

Conventions for arguments passing The purpose is to give a precise specification of the interface between the evaluator and a C-routine. This is a crucial point since the user writing his/her own C-routine to be integrated with existing library, must supply the arguments in a well defined format in order to be handled properly by the evaluator. Conversely, an argument area supplied by the evaluator must be correctly interpreted by a C-routine unless something could go wrong.
Since a C-routine may take a variable number of arguments, rather than supplying a list of arguments, a pointer to a memory area containing the list of arguments is supplied instead. As a result of this choice, two points have to be specified:

- how ASAX data types (**integer** and **string**) are actually represented;
- how these data are organized in the arguments area so they can be parsed correctly by a C-routine.

ASAX data representation: The integer type in ASAX is simply represented as a 4-bytes **int** in C since the evaluator was written in C. As a result, since ASAX code is supposed to be completely portable, the user bears the responsibility to choose the the appropriate scalar type (**short**, **int**, **long int**) having 4 bytes in size. This integer-size assumption is necessary in order to avoid problems when porting ASAX to an architecture with a different word size characteristics.
A string s of length l is represented by a pointer to a memory area of two fields. The first one which is 2-bytes long is the length of s and the second is the string itself which is 2*((l+1) div 2) bytes long. This area is always aligned on 2 bytes boundaries.

Format of an argument area: A predefined C-procedure $p(p_1,\ldots,p_n)$ is implemented by a C-routine which takes a single argument which is the address of a memory area representing the argument list. This area is always aligned on a 4 bytes boundaries for compatibility reasons and has two different formats depending on the kind of the predefined routine: fixed or variable-length.

1. Fixed arity: (see Figure 3.26)
   Consider a given argument p_i. If p_i is a value argument:

   - of type **integer**, p_i is this integer;
   - of type **string**, p_i is the pointer representing this string.

   if p_i is a reference argument:

   - of type **integer**, p_i is a 4-bytes pointer to a 4-bytes **int**;
   - of type **string**, p_i is a 4-bytes pointer to a 4-bytes aligned area.

   The example of Figure 3.27 illustrates the four above cases. So the internal code generated for it is shown in Figure 3.28

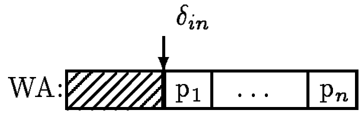

Figure 3.26: layout of an argument area

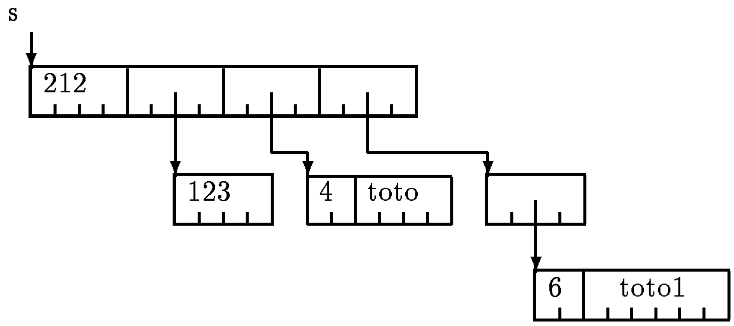

Figure 3.27: example of an argument area

2. Variable arity: (see Figure 3.29)

   The first field of this area is a 4-bytes **int** which is the actual number of arguments passed to the predefined routine and then a contiguous sequence of pairs (t_i, v_i) where t_i is the type of the i-th argument and v_i its value whose representation follows the same conventions as for fixed arity routines.
   So the internal code generated for it is shown in Figure 3.30. It is similar to that of fixed arity except that the actual arity as well as the type of each argument is determined at compile time. The boxes ra(expr$_i$, -,-) have the same content as for the fixed arity case. The first box stores the actual arity at the beginning of the argument area; the other boxes store the type of each argument right before its value.

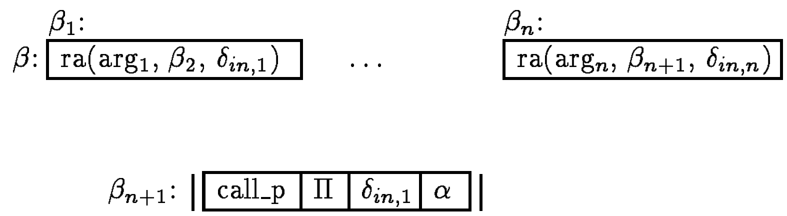

Figure 3.28: Predefined procedure call box notation

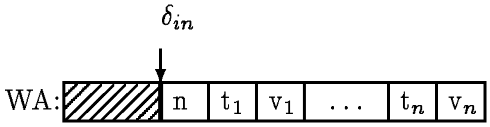

Figure 3.29: layout of an argument area

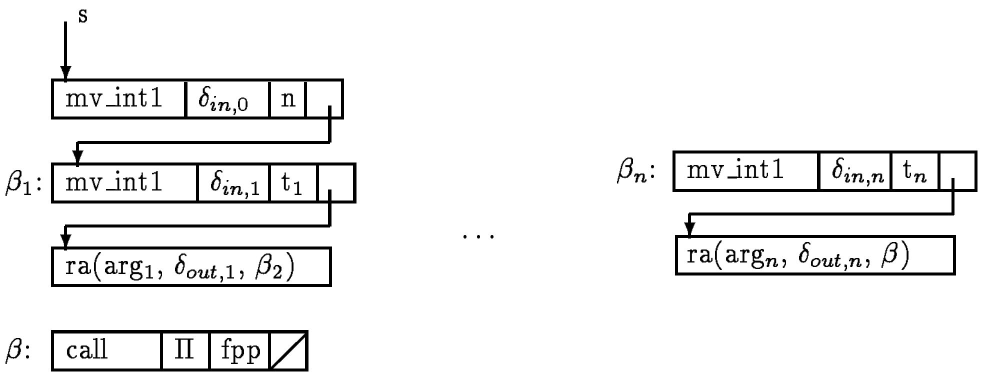

Figure 3.30: Reference parameter

For both cases (fixed and variable-length argument list), we have the following:

- $\delta_{in,1} = \delta_{in}$ if the routine is a procedure otherwise, $\delta_{in,1} = \delta_{in} + lv$ where *lv* is the length in bytes of an **int** and the returned value of the predefined function is stored at the address $\delta_{in,1}$;

- $\delta_{in,i+1} = \delta_{out,i}$;

- if arg_i is a value parameter, and no box is generated for its evaluation (i.e., arg_i is either a constant, a variable or a field name), ra(arg_i, $\delta_{out,i}$, β_{i+1}) is shown in Figure 3.31 where γ_i is the address of arg_i. The semantics of this box is move the integer value from address γ_i to the address $\delta_{out,i}$. Otherwise, ra(arg_i, β, δ) is simply ri(arg_i, δ, β).

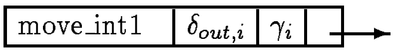

Figure 3.31: move box

- if arg_i is a reference parameter, ra(arg_i, β, δ) consists of the box given in Figure 3.32 where γ is the address of the variable arg_i. This abstract instruction simply moves γ at location δ.

assign_addr1	δ	γ	β

Figure 3.32: Reference parameter

- *call* $\in$ { *call_p1*, *call_f1*}.
  - *call_p1* if this is a call to a procedure;
  - *call_f1* if this is a call to a function.
- Π is the address of the routine p;

The value returned by a C-routine implementing a predefined function is the one returned by this function. This is always ensured by the evaluator.

3.4.9 Example of a rule internal code

Consider the following sample rule:

```
rule targetUser(userid, timelmt, flag:  string);
    begin
        if
         evt = 'login' and
         userid present and
         timestp > timelmt
                        → sendmsg(Msg1, uid, timestp);
         station = 'console'
                        → sendalarm(Msg2, uid, timestp);
        fi;
        flag := 'on';
    end
```

It is used to track a targeted user and sends a message $\texttt{Msg}_1$ if this user tries to log the system later than a certain time limit `timelmt`. It also sends an alarm message $\texttt{Msg}_2$ if this same user was logging the console. In either cases, this rule sets a certain flag to `'on'`.
The corresponding internal code is given in Figure 3.33.

As stated above, when the address `nil` is reached, the rule execution has been completed.

3.5 Abstract machine instruction set

This section is an overview of the abstract machine language. The abstract machine instructions (instruction set) are fully described by their parameters, their effect as well as the C declaration for their representation. The effect of an abstract machine instruction is explained without relating it to

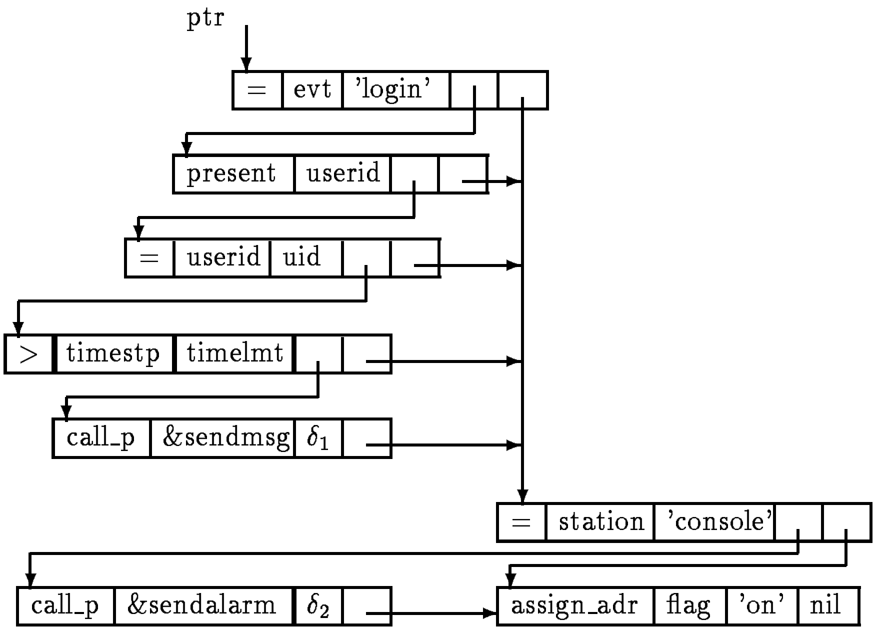

Figure 3.33: Rule internal code: Example

how the instruction contributes to the rule-based implementation.
Every abstract machine instruction (a.m.i) consists of an operation part followed by a number of arguments. The value of the operation part fully identifies the abstract instruction. These values can be divided into the following classes:

arithmetic arithmetic instructions;

relational relational instructions involving integers or strings;

assignment of integer or string variables;

audit data presence an instruction that simply checks if a given audit data is part of the current audit record;

rule triggering a set of abstract instructions that initialize and push a rule triggering descriptor in a stack of active rules;

pre-defined routine call instructions that call a pre-defined procedure or function;

which are described in the sequel.

3.5.1 arithmetic

The box notation of this a.m.i. is in Figure 3.34 where w denotes an arithmetic operator (*****, **div**, $+$, $-$) and γ_1 and γ_2 denote addresses of integer

$$\beta: \ \Vert\, \boxed{\omega \mid \delta_{in} \mid \gamma_1 \mid \gamma_2 \mid \alpha} \,\Vert$$

Figure 3.34: Arithmetic operator

values v_1, v_2.
The effect of this instruction is to store the value $v_1 \ \omega \ v_2$ at location δ_{in} and then to execute the a.m.i. at location α.

3.5.2 relational

The box notation of this a.m.i. is shown in Figure 3.5.2 where w denotes a

$$\beta: \ \Vert\, \boxed{\omega \mid \gamma_1 \mid \gamma_2 \mid \alpha_t \mid \alpha_f} \,\Vert$$

relational operator ($\{<, \leq, >, \geq, =, != \}$) and γ_1 and γ_2 denote addresses of integer or string values v_1, v_2. The effect of this instruction is to execute the a.m.i. at location α_t (resp. α_f) if $\mathrm{v}_1 \ w \ \mathrm{v}_2$ is evaluated to **true** (resp. **false**). There is 12 values of the operation part in this class depending on the relational sign involved and the type of the argument (integer or string).

3.5.3 assignment

This abstract machine instruction moves pieces of data between two locations. We distinguish two kinds of *assign* instructions depending on how the content of these locations is interpreted.

assign1

The box notation of this a.m.i. is in Figure 3.35

$$\beta: \ \Vert\, \boxed{\text{assign1} \mid \gamma_1 \mid \gamma_2 \mid \alpha} \,\Vert$$

Figure 3.35: integer assignment

where γ_2 is the address of an integer value. The effect of this instruction is to move the integer value at location γ_2 to location γ_1 and then to execute the abstract machine instruction at location α.

assign_str1

The box notation of this a.m.i. is given in Figure 3.36 where γ_2 is the address of a string. The effect of this instruction is to move the string from location γ_2 to location γ_1 and then to execute the abstract machine instruction at location α.

β:

assign_str1	γ_1	γ_2	α

Figure 3.36: string assignment

3.5.4 audit data presence

The box notation of this a.m.i. is given in Figure 3.37 where γ denotes a

β:

present	γ	α_t	α_f

Figure 3.37: Current record reference

string address. The effect of this a.m.i. is to execute the a.m.i. at location α_f if the length of the string is not 0 and to execute the a.m.i. at location α_t otherwise.

3.5.5 rule triggering

All boxes in this class assume the following global variables.

- *Size*
  This is an integer variable that eventually holds the size of the rule triggering descriptor. It is initialized to the size of the parameter area which is determined at compile time. At run time, it is incremented by the total size of the strings to be saved in the rule triggering descriptor;
- *Eff_ptr*
  This a pointer to the first free position in the parameter area of the rule triggering descriptor. When an argument value is copied to this area *Eff_ptr* is advanced by the required number of bytes;
- *Str_ptr*
  This is a pointer to the first free position in the area for saving strings in the rule triggering descriptor.

1. Set descriptor length

   (a) Box notation

setLen1	Size0	α

Figure 3.38: set length box

(b) arguments
int Size0;

(c) Effect:
Size = Size0;

(d) Note:
Size0 is the size of the working area after all rule arguments are evaluated;

2. Increment descriptor length

   (a) Box notation (see Figure 3.39)

addLen1	γ	α

Figure 3.39: add length box

   (b) arguments
   char **γ;

   (c) Effect:
   Size is incremented by the size of the string pointed by γ (in the format length, value);

   (d) Note:
   the incremented size is the size of the string plus the size of the length field of the string;

3. Push descriptor

   (a) Box notation (see Figure 3.40)

tr_mode	code	α

Figure 3.40: trigger box

   (b) arguments
   int tr_mode; /* identifier of the stack of active rules */
   Box *code; /* a pointer to the rule internal code */

   (c) Effect:
   a storage area is allocated for the rule triggering descriptor and pushed in the appropriate stack depending on the value of *tr_mode*. Eff_ptr and str_ptr are initialized as shown in Figure 3.41 ;

   (d) Note:
   the argument area as well as saved strings are copied in the rule triggering descriptor after the latter is pushed in the appropriate stack of active rules;

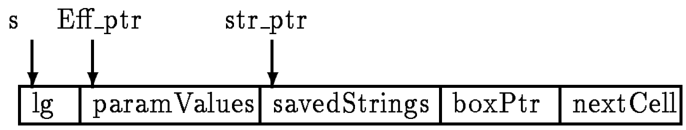

Figure 3.41: initialization of *Eff_ptr* and *str_ptr*

4. Copy integer

   (a) Box notation (see Figure 3.42)

cpy_int1	γ	α

   Figure 3.42: copy integer box

   (b) arguments
   int *γ;

   (c) Effect:
   the integer pointed by γ is copied at address *Eff_ptr*; *Eff_ptr* is incremented by the size of an integer;

   (d) Note:
   NONE;

5. Copy string address

   (a) Box notation (see Figure 3.43)

cpy_strAddr1	γ	α

   Figure 3.43: copy a string address box

   (b) arguments
   char **γ;

   (c) Effect:
   the string pointed by γ is copied at address *Eff_ptr*; *Eff_ptr* is incremented by the size of a string value;

   (d) Note:
   NONE;

6. Save string

   (a) Box notation (see Figure 3.44)

   (b) arguments
   char **γ;

save_str1	γ	α

Figure 3.44: save a string box

(c) Effect:
the value of *str_ptr* is assigned to *Eff_ptr*; *Eff_ptr* is incremented by the size of a string value, then the string itself is saved in the rule triggering descriptor starting from address *str_ptr*. *str_ptr* is incremented by the size of the saved string;

(d) Note:
NONE;

3.5.6 Pre-defined routine call

The box notation of this a.m.i. is in Figure 3.45

β: $\Big\|$ $\boxed{\text{call} \mid \Pi \mid \delta \mid \alpha}$ $\Big\|$

Figure 3.45: The *call* box

- if call = call_p1, this is interpreted as a call to a *procedure* where Π is its address. The effect is to call this procedure with the parameter list pointed to by δ and then to execute the abstract instruction at address α;

- if call = call_f1, this is interpreted as a call to a *function* where Π is its address. The effect is to call this function with the parameter list pointed to by δ + a size of an integer value. The returned value is stored at location δ. Then the abstract instruction at address α is executed.

assign address

This a.m.i is represented by the box in Figure 3.46 where γ_2 denotes a variable address. The effect is to assign this address to the variable pointed by γ_1 and then to execute the a.m.i pointed by α.

assign_addr1	γ_1	γ_2	α

Figure 3.46: assign address box

move integer box

This a.m.i is represented by the box in Figure 3.47 where *source* denotes an integer address. The effect is to move the integer from location *source* to location *dest* and then to execute the a.m.i pointed by α.

move_int1	dest	source	α

Figure 3.47: move integer box

move string box

This a.m.i is represented by the box in Figure 3.48 where *source* denotes a string address. The effect is to move the address of the string from location *source* to location *dest* and then to execute the a.m.i pointed by α.

move_str1	dest	source	α

Figure 3.48: move string box

move integer value box

This a.m.i is represented by the box in Figure 3.49 where *source* denotes an integer value. The effect is to store the integer v to the location δ and then to execute the a.m.i pointed by α.

mv_int1	δ	v	α

Figure 3.49: move integer value box

3.5.7 C declarations

The purpose here is to define a C type that holds all kinds of abstract machine instructions. This type is implemented by a structure having three fields. Two fields are common to all abstract machine instruction internal codes and the third one depends on each particular kind of abstract instruction. The common fields are the operator part and the address of the next instruction to be executed while the third part is interpreted depending on the operator part and is defined by a union. Let us first give the C declarations for the third part corresponding to each kind of abstract machine instruction.

```
typedef struct {
	int val;
	char *deltaIn;
} mv_int;

typedef struct {
	int *gamma1, *gamma2;
} assign_int;

typedef struct {
	hole *gamma1, *gamma2;
} assign_str;

typedef struct {
	char *gamma1, *gamma2;
} assign_addr;

typedef struct {
	int *deltaIn, *gamma;
} un_minus;

typedef struct {
	int *deltaIn, *gamma1, *gamma2;
} arithm;

typedef struct {
	void *(*fpptr )();
	char *deltaIn;
} call;

typedef struct {
	char **str_addr;
	BoxPtr alphaTrue;
} present_typ;

typedef struct {
	BoxPtr ruleCode;
} trigger;

typedef struct {
	char **gamma1, **gamma2;
	BoxPtr alphaTrue;
} StrRelat;

typedef struct {
	int *gamma1, *gamma2;
```

```
    BoxPtr alphaTrue;
} relational;

typedef struct {
    hole *dest, *source;
} move;

typedef struct ami {
    enum operation op;
    union {
        mv_int          mv_intbox;
        assign_int      assign_intbox;
        assign_str      assign_strbox;
        assign_addr     assign_addrbox;
        un_minus        un_minusbox;
        arithm          arithmbox;
        StrRelat        StrRelatbox;
        relational      relationalbox;
        present_typ     presentbox;
        trigger         triggerbox;
        char            *rule_name;
        call            callbox;
        move            movebox;
        hole            *gamma;
        int             size;
    } arg;
    BoxPtr alpha;
} Box;
```

Chapter 4

Overview of the syntax analyser implementation

4.1 Introduction

This part of the document is devoted to the implementation design of the analyser. The basic functions achieved by the analyser are the following:

- detect syntactic and semantic errors occurring in the program text and help the user finding and correcting them;
- in case the program is error free, the analyser has to generate an internal code for the program. This internal code will be used by the interpreter in order to execute the program;

Therefore, the main issues discussed here are:

- the lexical analyser or scanner which offers a set of functions that transform a program text into a set of words or tokens;
- The syntactic and semantic analyser which provides parsing functions as well as the primary type checking routines;
- the global data structures as well as the functions handling them.

4.2 Analysis principles

In this section we discuss informally the implementation design principles and main ideas. In the subsequent sections, we examine in detail the various steps of the analysis.
The scanner implementation is concerned with recognizing tokens in the program text to be analysed and replacing them with a more convenient representation. Tokens are recognized according to their BNF syntax given in 4.4.3. The last symbol being read is stored in a global variable along with its type (identifier, integer, C_string,... etc). The main ideas of its implementation is that the scanner is always one character ahead. At the end of

the lexical analysis, the program text is converted from a row sequence of characters to a sequence of symbols.

The syntax analyser or parser task involves recognizing correct sentences and detecting possible syntax errors. The parser deals with the symbol sequence built up by the scanner. When the parser is executed it examines one or more symbols and determines whether they form a syntactically legal sentence as described by the language syntax. If the program is error free, internal code is generated for it. This code is later used to execute the program appropriately. The parser is always one symbol ahead i.e., it uses a single-symbol look-ahead method.
The parser uses the external variable *symb* which holds the internal representation of the last symbol being read. This symbol along with the remaining symbol sequence in the program file to be analysed forms what we call the *current symbol sequence* that is the sequence of symbols to be treated next and will be denoted by *css* throughout. For every BNF rule, the parser contains a subprogram that is dedicated for parsing constructs having such a BNF grammar. When such a subprogram is activated, the *css* has (if error free) a prefix that satisfies this BNF grammar. However, the *css* may have many such a prefix. It turns out that the prefix to be considered is in fact the greatest one.

Besides syntax checking, the parser has to generate internal code of all constructs it recognizes. An important feature of the analysis discussed here is that it is performed in a single pass. Consequently, the generated internal code is the final one and no intermediate code is needed. The resulting internal code is ready to use by the interpreter. To achieve this feature, the parser uses a technique consisting of generating incomplete codes. The main idea behind this is that at the moment the parser generates an internal code, it does not know the value of certain data fields yet. For instance, the parser does not know where certain pointers must go. The point is that the parser will keep track of these missing data fields until their values are available. We will examine this in more details in the next section.

Among the other functions of the parser, some are used (as auxiliary functions) to manage certain global data structures such as:

- the current record table which is used to access the current record;
- the rule descriptors table that contains a descriptor for each rule in the program;
- external to internal audit data identifier mapping table.

4.3 Main global data structures

4.3.1 Rule descriptors table

The set of rules declared in a file are represented by three tables. The first one *trdescr*[] is an array rule descriptors, the second *tdescr*[] is an array of formal parameter and local variable descriptors and the third *targ_types*[] is an array of argument types. A rule descriptor contains the necessary information relevant for a rule declaration. When the parser recognizes a rule declaration, it collects all information about this rule (name, arguments, local variables, action part, ..., etc), adds a new descriptor for this rule to *trdescr*[] and adds to *tdescr*[] as many parameter and local variable descriptors as there is in this rule declaration.
A rule descriptor is a structure of the following fields:

- rule name;
- a pointer to its internal code;
- number of formal parameters;
- number of local variables;
- the size in bytes of the Actual Parameter Area (A.P.A);
- rule status which is a flag that indicates whether the rule was:
  - not declared yet and is triggered for the first time or;
  - not declared yet and was triggered at least once;
  - already declared.
- index in *tdescr*[] of the first formal parameter or local variable descriptor related to this rule. The value of this field is undefined if parser had not yet encountered a declaration for this rule. In this case, the next field is considered instead;
- index in *targ_types*[] of the first argument type related to this rule.

If a rule declaration was already encountered in the source file, arity and type checking are performed using the table *tdescr*[] otherwise, the table *targ_types*[] is used instead. The parser evaluates the *status* field to determine which table (*tdescr*[] or *targ_types*[]) to consult. An entry in *tdescr*[] is a structure of the following fields:

- a flag that indicates if this is a formal parameter or a local variable;
- its name;
- its type;

An entry in *targ_types*[] is simply the type of an argument in a rule triggering. These three tables must be declared external to at least two functions:

- functions that parse a rule declaration and creates a descriptor for it;
- functions which parse a rule call and must check if this call is done appropriately i.e. correct parameter number and types (type checking will be addressed in a later section).

The descriptors in *trdescr*[], *tdescr*[] and *targ_types*[] are stored in the order they occur in the source file.
Here is the C declarations of these tables:

```
typedef struct {
     char  rule_name[ident_lg];
     Box   *Code;      /* internal code of this rule      */
     int   ParNre;     /* nbre of formal parameters       */
     int   VarNre;     /* nbre of local variables         */
     int   APASize;    /* size in bytes of A.P.A.         */
     int   status;     /* unknown, known or declared rule */
     int   nameDescr;  /* index into tdescr[]             */
     int   targDescr;  /* index into targ_types[]         */
} rdescr;

rdescr trdescr[MaxRuleNr];

enum type {integer, bytestring, fieldstring, undef};
enum var_par {var, par};

typedef struct {
     enum var_par  iname;           /* param. or a var.  */
     char          name[ident_lg];  /* its name          */
     enum type     tname;           /* its type          */
} namedescr;

namedescr tdescr[MaxVarNr];

enum type targ_types[MaxVarNr];
```

4.3.2 Current record table

The current record table must be visible to all the parsing functions because field names may occur everywhere in a program:

- as argument of a rule triggering or a procedure call;
- as a subexpression of any expression.

As mentioned in the previous section, all references to a current record field in a rule declaration is converted into an indirection through the current record table. Consequently, this table must be declared external to the following functions:

- all parsing functions;
- all subprograms that create internal codes;
- all functions that initialize and update this table.

This last category of functions add new entries in the indirection table as new references to audit data are encountered during the parsing process. At the end of the parsing, all references to the current audit record will have been registered in the indirection table.
Furthermore, because this table must be sorted with respect to audit data identifiers, a newly encountered field name is inserted in this table so as this order is maintained after insertion.
As a result, this table is implemented as an array where each entry is related to an audit data and have the following items:

- the audit data identifier;
- a pointer to this audit data inside the current record;

Finally, here is the C declaration of a current record table entry:

```
typedef struct {
        unsigned short ad_id;     /* audit data id    */
        char           *ad_value; /* audit data value */
} CR_table_entry;

CR_table_entry CR_table[Max_ad_id];
```

A field name reference contained in a construct internal code (for instance an expression involving the current record fields) is then replaced by the address of the indirection table entry that contains the pointer to the field inside the current record i.e., the address of the field *ad_value* in this table. This address is fully determined at the end of the analysis. (See Figure 4.1). Rules for accessing and updating this table are described in more details in the next sections.

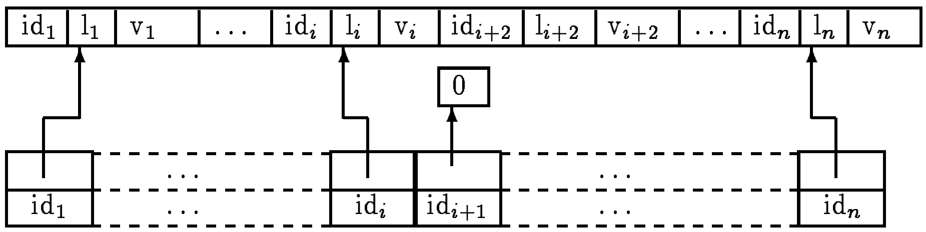

Figure 4.1: The current record table

Note

It is interesting to note that although the physical location of a particular audit data can vary from an audit record to another, the indirection table allows direct and constant access to audit data.

4.3.3 Standard library table

The standard library table must be declared external to the function that parses predefined procedure calls. It is used to make the necessary checks to make sure the call is done correctly i.e:

- there exist a standard library table entry corresponding to this call;
- the arguments are used accordingly i.e:
  - same arity (number of arguments);
  - argument type (integer or string of bytes);
  - argument passing type (call by reference or by value);

See the previous chapter for the C declaration of the standard library table.

4.3.4 Holes and list of holes

The problem of incomplete internal representation was outlined in a previous section. To examine this in more details, let us introduce some definitions. A *hole* is simply a location that has not yet been assigned a value. Suppose we know for some reasons that a set of holes have to be assigned the same value at a certain point during execution. To keep track of all these holes, we form what we call a *list of holes* which is simply a linked list where each hole is assigned the location of the next hole in the list; the last hole been simply non instantiated. As soon as the missing value is obtained (as a return value of a given routine or whatever), it is easy to scan this list and assign each of its holes this given value.

The problem of incomplete internal code is based on managing holes and lists of holes. The box notation of an internal code always contains the location of the next statement to be executed. Therefore the parser must generate an internal code for a given statement although it lacks the location of this next statement. For example, the corresponding box notation for the condition:

present user_id **or present** process_id

is internally coded as shown in Figure 4.2.
At the moment the subexpression:

present user_id

is parsed and an internal code generated for it, there is no internal code for the second subexpression:

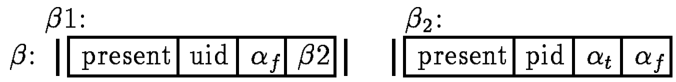

Figure 4.2: Box notation

present process_id

and therefore, the address of this internal code is not known. Clearly, β_2 is a hole.

In general, the internal code of a given construct *cons* is a set of one or more (terminal) boxes containing possibly some holes. To cope with this situation, we associate to each construct internal representation one or more lists of holes. All holes of one such a list must later be assigned the same address as soon as this address is known. As the parsing proceeds, new lists of holes are created and others are instantiated. In fact, either a construct is a condition and we have exactly two lists of holes one related to the branch address α_t and the other to the branch address α_f. Or the construct is not a condition and in this case a unique list of holes is associated to the corresponding internal code. This list must be assigned the location of the next statement to be executed. Let us see this in examples.

Example 1

Consider the following condition:

a **or** b **or** c

where a, b and c denotes conditions. The box notation of the corresponding internal code is of the form:

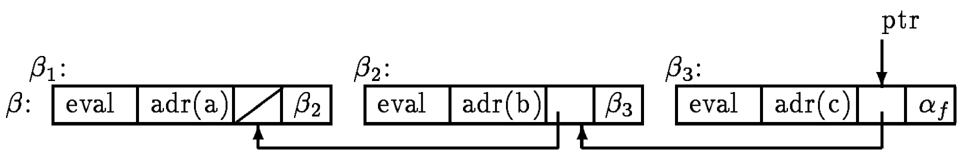

Figure 4.3: Example of list of holes

Clearly, we have two lists of holes. The first one is related to α_t and the other to α_f. Note that an elementary condition has two lists of holes of one element each.

Example 2

Consider the conditional action:

$$\textbf{if } b_1 \rightarrow a_1; \ldots; b_n \rightarrow a_n \textbf{ fi}$$

where b_1, ..., b_n denotes conditions and a_1, ..., a_n denotes actions. The corresponding internal code is of the form:

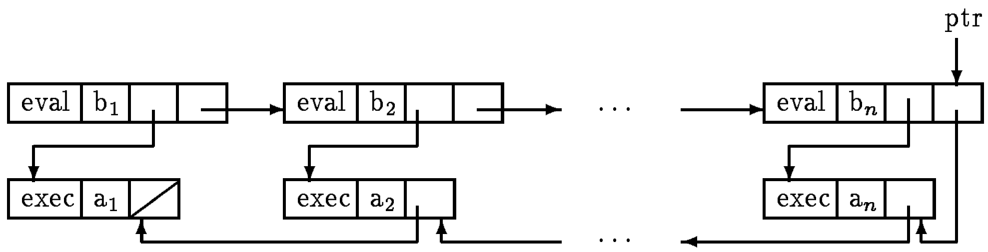

Figure 4.4: Conditional action incomplete internal code

which has a list of holes of n+1 elements. The location of the next statement to be executed after the **if fi** statement is related to this list.

Similarly, references to the current audit record cannot be determined until all the rules are parsed. When a reference to a particular audit data is encountered during the parsing, it is inserted as a member of a list of holes related to this reference. At the end of the analysis, when all the the current audit record references are known and registered in the current record table, all holes belonging to a particular list of holes are updated to point to the related current record table entry.

Note that the implementation of lists of holes uses these very holes to keep track of them and no additional auxiliary data structures are used to store them.

At any time of the parsing we have either one or two lists of holes. These lists must be external to all parsing functions because a list of holes created (or extended) by a given function must be passed to the parsing function handling the next statement.

4.3.5 Names mapping table

For the sake of efficient condition evaluation, field names are coded into internal form which is simply the audit data identifier. This table is a mapping between external and internal field names.
Given an external field name, we can search this table to get the corresponding field name identifier. Note that this table is based on the *audit data description file* generated by the format adaptor (see [6]) and must be

external to all parsing functions. Here is the C declaration of such a table:

```
typedef struct {
     unsigned short ad_id;         /* audit data id   */
     char           *field_name;   /* audit data name */
} map_slot;

map_slot mapping_table[Max_ad_id];
```

4.4 The lexical analyser

4.4.1 Introduction

In this section, the scanner is examined and specified in detail. After recalling the rule-based language concrete syntax, we will discuss the scanning principle, the global data structures and related functions as well as the specification of the main scanning functions. Here is the concrete syntax of a token:

⟨token ⟩ ::= ⟨identifier ⟩
    | ⟨constant ⟩
    | ⟨special symbol ⟩

⟨identifier ⟩ ::= ⟨letter ⟩
    | ⟨identifier ⟩ ⟨digit ⟩
    | ⟨identifier ⟩ ⟨letter ⟩
    | ⟨identifier ⟩ ⟨underscore ⟩ ⟨letter ⟩
    | ⟨identifier ⟩ ⟨underscore ⟩ ⟨digit ⟩

⟨constant ⟩ ::= ⟨integer constant ⟩
    | ⟨C_literal ⟩
    | ⟨X_literal ⟩

⟨integer constant ⟩ ::= ⟨digit ⟩
    | ⟨integer constant ⟩ ⟨digit ⟩

⟨C_literal ⟩ ::=' ⟨C_sequence ⟩'

⟨C_sequence ⟩ ::= ⟨empty ⟩
    | ⟨C_sequence ⟩ ⟨C_character ⟩

⟨C_character ⟩ ::= any printable ASCII character except simple
    quote (') | ' '

⟨X_literal ⟩ ::= X ' ⟨X_sequence ⟩'

⟨X_sequence ⟩ ::= ⟨empty ⟩
| ⟨X_sequence ⟩ ⟨X_symbol ⟩

⟨X_symbol ⟩ ::= ⟨X_character ⟩ ⟨X_character ⟩

⟨X_character ⟩ ::= 0 | ... | 9 |A |B |C |D |E |F

⟨digit ⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

⟨letter ⟩ ::= a | ... | z | A | ... | Z

⟨special symbol ⟩ ::= + | − | * | (|) | > | < | != | ≤ | | ≥ | :
| %= | ; | = | -> | := | , | **and** | **at_completion**
| **begin** | **div** | **do** | **end** | **false** | **fi** | **for_current**
| **for_next** | **not** | **od** | **off** | **or** | **present** | **rule** | **skip**
| **string** | **if** | **integer** | **mod** | **trigger** | **true** | **var**

4.4.2 Lexical analysis principle

The scanner recognizes the tokens in accordance with the above syntax and returns for each token an internal code which will be described here. The scanner is always one character ahead i.e., at the time it returns a token, it has already read the character ahead of this token.

4.4.3 Specification of the scanner

The scanner *getsymb* reads the next token in the standard input (stdin). It has the following external objects:

- **specsymbtab**[]
  which is a table of all special symbols stored as strings:
  char * **specsymbtab**[]={"and","at_completion","begin"
  ,"div","do","end","false","fi","for_current",
  "for_next","if","integer","mod","not",
  "od","off","or","present","rule","skip","string",
  "trigger","true","var"};

- *symb* which holds the last read symbol. Its type is defined as follows:

```
typedef struct symb {
        enum index ibsymb;  /* index in specsymbtab[]      */
        int X_lg;           /* length of X_literal         */
        union bsymbval {
        char svalue[MAXS];  /* identifier, spec.  symbol */
                            /* C or X_literal;            */
        int ivalue          /* an integer or a boolean    */
```

```
        }
};
```

The variable *ibsymb* indicates the type of the token which is defined by the set {identifier, special_symbol, X_literal, C_literal, integer, boolean}. The second member of the structure is the token itself and is represented by a union.

- *nextchar* the last character been read;

A scanner call has the following effect:
Let α be the current input stream. α can be uniquely decomposed into the form:

$$\alpha'\beta$$

where α' is a possibly empty sequence of spaces and β a possibly empty character sequence not beginning with a space character.
If there exists a greatest prefix σ of β such that:

- σ is a token;
- β is of the form $\sigma\gamma$;

The effect of *getsymb* is to assign the internal representation of σ to *symb*; after this call, the content of the input stream is γ; the first character of γ been returned in *nextchar*.
If β is of the form $?\beta'$, the effect of the call is to assign the value *nosymb* to `symb.ibsymb`; the content of stdin been β'.
otherwise, *getsymb* assign `symb.ibsymb` the value *errsymb* leaving the input stream in an undefined state.

The scanner module contains the following main functions:

- `get_char;`
- `skipspaces;`
- `get_identifier;`
- `get_Cstring;`
- `get_Xstring;`
- `get_integer;`

which are here under specified:

4.4.4 Auxiliary functions specification

Each of the above functions has the following external objects:

- `specsymbtab[];`
- `symb;`
- `nextchar;`
- `morechar;`

Let c = nextchar i.e., the last character been read and α the current input stream; in the following, cα is called the *current character sequence*.

get_char

precondition
the current input stream is a possibly empty character sequence α;

postcondition
If α is empty:

- a special character '?' is returned in nextchar;
- morechar is set to 0.

Otherwise, α = cα'

- the first character of α is returned in nextchar;
- morechar is set to 1;
- the current input stream is α'.

skipspaces

precondition
the current input stream is a possibly empty character sequence α;

postcondition

- If α is empty, skipspaces has no effect;
- Otherwise, α is of the form $\alpha'\beta$ where α' is a possibly empty sequence of spaces or end of lines and β is a possibly empty sequence of characters not beginning with a space or an end of line character;
- the effect of `skipspaces` is to put the current character sequence in the form β.

get_identifier

precondition

the current character sequence is cα where c is a letter and α is a possibly empty character sequence;

postcondition

- c$\alpha = \sigma\gamma$ where σ is the greatest prefix of cα such that σ corresponds to an identifier;
- the effect of get_identifier is to return the internal representation of σ in *symb* and set the current character sequence to be γ.

get_integer

precondition

the current character sequence is cα where

- c is a decimal character;
- α is a possibly empty character sequence.

postcondition

- c$\alpha = \sigma\gamma$ where
  - σ is the greatest prefix of cα such that σ is a sequence of decimal characters representing an integer constant;
  - γ is a possibly empty character sequence.
- the effect of get_identifier is to return in *symb* the internal representation of σ and put the current character sequence to be γ.

get_C_literal

precondition

the current character sequence is cα where

- c is ';
- α is a possibly empty character sequence.

postcondition

- if c$\alpha = {}'\beta'\gamma$ where
  - β is a C sequence (see 4.4.1);
  - γ is a possibly empty character sequence.
  - the effect of **get_C_literal** is to return in *symb* the internal representation of the C literal represented by β and put the current character sequence to be γ;
- otherwise, the value *errsymb* is returned in `symb.ibsymb`.

4.5 Functions handling the global data structures

4.5.1 add_rule_descriptor

The rule descriptor table *trdescr*[] is a set of rule descriptors

$$rdescr_1, \ldots, rdescr_n \qquad (n \geq 0)$$

where

$$rdescr_i = (\text{rule\_name}_i, \text{for\_par}_i, \text{loc\_var}_i, \text{action\_part}_i) \qquad (i = 1, ..., n).$$

As a result, a rule descriptor table can be defined by the corresponding set of rule descriptors.

precondition

- RD_0 is a rule descriptor table associated to the set of rule descriptors $\{rdecr_1, \ldots, rdecr_n\}$ $(n \geq 0)$;
- *rdecr* = (rule_name, for_par, loc_var, action_part) where
  - rule_name is a pointer to the rule name internal code;
  - for_par is a pointer to a formal parameter descriptor;
  - loc_var is a pointer to a local variable descriptor;
  - action_part is a pointer to the internal code of the rule rule_name action part.

postcondition
the effect of this function is to update the rule descriptor table *trdescr*[] so that it corresponds to the set $RD_1 = RD_0 \cup \{rdecr\}$.

4.5.2 add_field_name

This function adds a new entry to the current record table. Recall that this table is implemented as a linked list of cells each one related to an audit data.
The current record table is defined by the corresponding set of cells:

$$\{cell_1, \ldots, cell_n\} \qquad (n \geq 0).$$

A cell is a triple:

- audit data identifier;
- pointer inside the current record;
- pointer to the next cell.

This set is sorted with ascending audit data identifiers.

precondition

- CRT_0 is the current record table defined by the corresponding set of cells $\mathrm{C}_0 = \{cell_1, \ldots, cell_n\}$ $(\mathrm{n} \geq 0)$. This set is sorted with ascending audit data identifiers;
- *id* is an audit data identifier.

postcondition

the effect of this function is to return a pointer to a current record table defined by the set of cells

$$\mathrm{C}_1 = \mathrm{C}_0 \cup \{cell\}$$

where *cell* have *id* as its audit data identifier.

4.5.3 get_audit_data_id

The purpose of this function is to search the table of external into internal field names mapping for the audit data identifier corresponding to a given audit data external name.

global objects

table of external into internal field name mapping;

precondition

- the names mapping table is instantiated;
- it is sorted with external names;
- *field_name* is a string;

postcondition

- the names mapping table is unchanged;
- if there exists a table entry (external_name$_i$, audit_data_id$_i$) such that field_name = external_name$_i$, a pointer to id$_i$ is returned;
- otherwise, the value -1 is returned.

4.5.4 insert_incomplete

The purpose here is simply to add a new hole to a list of holes.

precondition

- `hole_list_ptr` is a pointer to a list of holes;
- `hole_ptr` is a pointer to a hole i.e., the address of an non instantiated pointer;

postcondition

- the hole pointed to by `hole_ptr` is added at the beginning of the hole list pointed to by `hole_list_ptr`.

4.5.5 concate_incomplete

Given two lists of holes, the effect of this function is to concatenate them and return a pointer to the resulting list of holes.

precondition

`hole_list_ptr`$_1$ and `hole_list_ptr`$_2$ are two hole lists pointers;

postcondition

- let `last_hole` be the last hole of the hole list pointed to by `hole_list_ptr`$_1$;
- The effect of this function is make this pointer point to the second list i.e.,`last_hole` = `hole_list_ptr`$_2$ and return `hole_list_ptr`$_1$.

4.6 Specification of the parser functions

4.6.1 Common features

Before giving the specification of the main parsing functions, it would be useful to set up a terminology that will allow us to provide classes of specifications instead of a specification for each particular function.
For each syntactical category (condition, expression, action, rule, ...etc), a parsing function is provided for its handling. A table giving the set of syntactical categories and the corresponding functions is given below.

1. let γ be the current character sequence, (see 4.4.4). Suppose γ has a greatest prefix σ such that σ is a sequence of legal symbols

   $$s_2,\ \ldots,\ s_n \qquad (\mathrm{n} \geq 0)$$

   Suppose the last symbol been read is s_1 (returned in the external variable *symb*). The symbol sequence $s_1,\ s_2,\ \ldots,\ s_n$ is called the *current symbol sequence*. This means we are always one symbol ahead.

2. consider a function corresponding to a given syntactical category and let

   - $s_1,\ \ldots,\ s_n \quad (\mathrm{n} \geq 1)$ be the current symbol sequence at a given time of execution;
   - $s_1,\ \ldots,\ s_j \quad (1 \leq \mathrm{j} \leq \mathrm{n})$ be a non empty prefix of this sequence.

we say that at that moment this function has the effect of *handling* the symbol sequence $s_1,\ \ldots,\ s_j$ if $s_{j+1},\ \ldots,\ s_n$ is the current symbol sequence *after* call to this function and if the following conditions hold:

1. let

   - γ be the current character sequence *before* the call;

- α be the greatest prefix of γ corresponding to the symbol sequence $s_2, \ldots, s_n$.

then γ is of the form $\alpha\beta$ where β is an arbitrary character sequence.

- **if j $\leq$ n**, the current character sequence after execution is of the form $\alpha'\beta$ where α' is the character sequence corresponding to the symbol sequence $s_{j+2}, \ldots, s_n$;
- **if j $=$ n**, and β is of the form $?\beta'$, after execution, the current input stream is β' and `symb.ibsymb = nosymb`;
- **if j $=$ n** and β is not of the form $?\beta'$, the execution is undetermined and `symb.ibsymb = errsymb`.

2. in the first two cases (j < n and j = n), we have the following conditions:

   (a) i. in absence of syntactic or semantics errors (a list of semantics errors will be given later), the function returns a pointer to the internal code corresponding to the language construct represented by the sequence $s_1, \ldots, s_j$;

   ii. for every field name occurring in $s_1, \ldots, s_j$, the current record table is updated to contain a new entry (if this reference has not been previously encountered.) for this audit data;

   iii. a pointer to the list of holes (two lists of holes in case of conditions) corresponding to the generated incomplete internal code;

   iv. in case that $s_1, \ldots, s_j$ represents a rule declaration, the table of rule descriptors is updated to include an additional entry corresponding to this rule descriptor;

   (b) in case of errors (syntactic or semantics) an error message is sent. Every error is identified by a code which is a character string.

Now, we are in position to give the specification of parsing functions.

4.6.2 Parsing functions specification

These functions are local to the syntactic and semantic module, they are declared in an environment containing the following external objects:

- *symb* of type *bsymb* (see 4.4.3);
- *sxerror* a boolean variable which is set to 1 if an error has been encountered and to 0 otherwise;
- the declaration of the external objects described so far (rule descriptor table, current record table, table of names mapping, ..., etc);
- the definition of the needed types such as:
  - *bsymb*;

– internal codes data structure.

The specification of such functions is then the following:
Let

- *sxcat* be the name of such a function;
- *cat* the corresponding syntactic category;
- S the current symbol sequence before call to *sxcat*:

If there exist a greatest prefix C of S representing a construct belonging to the syntactical category *cat*, the effect of *sxcat* is to *handle* this prefix. Otherwise, the call to *sxcatr* generates one or more error messages describing the encountered error.

Tables of correspondence

sxrule	rule declaration
sxlparam	parameter group
sxlocal_var	variable declaration part
sxaction	action part
sxcondition	condition
em sxcat	*cat*

The function sxaction has the following auxiliary functions each one corresponding to a particular item of the action BNF syntax:

sxskip	skip action
sxassign	assignment
sxconditional	conditional action
sxrepetitive	repetitive action
sxcompound	compound action
sxrule_trig	rule triggering
sxcall	predefined procedure call

Similarly, sxcondition has the following auxiliary functions:

sxconj	conjunction
sxsimple_cond	simple condition
sxtrue	**true**
sxfalse	**false**
sxpresent	**present** <field_name>
sxrelational	relational expression
sxnegation	negation

Appendix A

System parameters

This appendix describes some limit figures of the system parameters. An error message is output whenever one of these limits is exceeded in which case the execution is aborted. This can occur either during the parsing of the source file (compile time) or during the audit file analysis process (run time). The columns of the following table are hereunder described:

name the name of the system parameter for which the limit is imposed;

max. the maximum value that must not be exceeded;

description an explanation of what is represented by this parameter;

line the line number of the source line in the source file *ASAX_param.h*. This make it possible to easily maintain system parameter values.

name	max.	description	line
MAXS	100	length of a C or X literal	1
ident_lg[1]	20	length of an identifier	2
MaxRuleNr	50	number of rule declarations	3
MaxVarNr[2]	150	total number of par. or var.	4
MaxVarRuleNr	30	number of var./rule declaration	5
MaxParRuleNr	30	number of par./rule declaration	6
Ext_Int_Nre	50	number of global var.	7
MAX_files	20	number of opened NADF files	8
Max_ad_id	1200	audit data ids	9
lgIntConst	200	number of integer constants	10
lgStrConst	200	number of string constants	11
NbreBoxes	1000	number of generated boxes	12

Notes

1. *ident_lg* is the maximum number of initial characters in an identifier that are significant;

2. *MaxVarNr* is the maximum number of local variables and formal parameters that can be declared in all rule declarations of a source file while *MaxVarRuleNr* and *MaxParRuleNr* are respectively the maximum number of variables and parameters that can be declared in a given rule declaration;

3. the type **integer** supported by the evaluator is assumed to be a 4-bytes **int**;

Appendix B

Compiler error messages

The ASAX compiler diagnostic messages fall into three classes: lexical, syntactic or semantic errors.

BoxPtrArray table overflow
: Indicates that the maximum number of abstract instructions has been reached. In this case, the constant `NbreBoxes` mentionned in the previous Annex must be set to a higher value;

Severe error: duplicate audit data
: The named field name in *audit data description file* has two distinct audit data ids;

action expected
: an action was expected. This error is often generated when the last action in a compound, repetitive or conditional action is semicolon terminated;

audit data identifiers must be $<$ 65536
: Audit data ids are represented as **unsigned short**, this maximum value is assumed for 2-bytes shorts;

check arity
: A rule or predefined function or procedure was called with a wrong number of arguments. The rule triggering, or the procedure or function call must be performed in accordance with the rule declaration or predefined procedure or function specification respectively;

error in expression
: An error occurred during compilation of an arithmetic expression. This can be caused for example by a missing operator or operand;

function not a procedure!
: A function call was used as a procedure call. The returned value of a function must be used as an expression;

identifier expected

An identifier was expected here but not found. An identifier is expected after the reserved word **rule**, as a left side of an assignment action and as a passed by reference parameter in a predefined function or procedure call;

invalid line

The source line given by its line number in the *audit data description file* is invalid. See audit data description file syntax;

mapping_table overflow

Indicates that the maximum number of field names was exceeded. In this case, the constant `Max_ad_id` mentionned in the previous Annex must be set to a higher value;

not a field name

The token after the reserved word **present** is not a legal field name. See your *audit data description file* to have the list of legal field names;

not a left value

The left hand side of assignment operator as well as a passed by reference parameter to a pre-defined routine must be the name of a local variable. A passed by reference parameter is not allowed to be a field name (nor a formal parameter of the rule) because this means the audit file could be modified by the analyser;

out of memory

The total working storage is exhausted. Compile the file on a machine with more memory or simplify the source file;

procedure not a function!

A procedure call was used as a function call: The procedure does not return a value;

redeclared rule

the source file contains at least two rule declarations with the same rule name;

semicolon expected

A semicolon was expected;

table of integer constants overflow

Indicates that the maximum number of integer constants in the program text was exceeded. In this case, the constant `lgIntConst` mentionned in the previous Annex must be set to a higher value;

table of string constants overflow

Indicates that the maximum number of string constants in the program text was exceeded. In this case, the constant `lgStrConst` mentionned in the previous Annex must be set to a higher value;

too long line

A line typed 5 in the *audit data description file* must not be more than 256 characters long. If needed, such a line could be broken into several consecutive lines of type 5;

type mismatch

The compiler detected an expression having a wrong type. This can occur if the sides of an assignment operator are not the same type or if the actual parameter (of a rule triggering or a pre-defined routine call) is not the same type as the corresponding formal parameter;

type name expected

A name other than **integer** and **string** was used as a type name in a local variables or formal parameters group declaration;

undefined function or procedure

The named function or procedure is not a part of the pre-defined library. This could be caused by a misspelling either at this point or at the declaration of this function or procedure as part of the pre-defined library;

undefined rule

The named rule has no declaration. This could be caused by a misspelling either at this point or at the declaration;

unknown identifier

The named identifier has no declaration. This could be caused by a misspelling at this point or the declaration. An identifier must be a local variable name, a parameter name or a field name;

warning: too long name

The identifier in lines typed 2, 3, or 4 of the *audit data description file* is more than 30 characters long. This name is truncated to its first 30 characters;

Bibliography

[1] N.Habra, B. Le Charlier, A. Mounji. *Preliminary report on Advanced Security Audit Trail Analysis on Unix* 15.12.91. 34 pages

[2] I. Mathieu. *Advanced Security Audit Analyser on Unix* 13.09.91. 40 pages

[3] F. Libion. *Towards "intelligent" Security Audit Trail Analysis Tools.*. MS Thesis. FUNDP Namur 1990-91. 130 pages.

[4] A. Baur, W. Weib. *Analysis of protocol data using AI techniques.* ZFE IS SOF 4. 07.11.89.

[5] SAT team. *Security Audit Trail step1.* 104.52.7 REV7. 22.11.91. 213 pages.

[6] E. Van Meerbeck, I. Mathieu. *SAT_X Format Adaptor* REV0 42.01.92. 35 pages.

[7] W. Kernighan, M. Ritchie. *The C programming language* 1978. 228 pages

[8] B. Le Charlier. *Language Simple Didactique (LSD/80)* 1980. 149 pages