NTIF: A General Symbolic Model for Communicating Sequential Processes with Data*

Hubert Garavel and Frédéric Lang

INRIA Rhône-Alpes / VASY
655, avenue de l'Europe
38330 Montbonnot Saint-Martin, France
{Hubert.Garavel, Frederic.Lang}@inria.fr

Abstract. One central problem in the computer-aided verification of concurrent systems consisting of communicating sequential processes with data is to find suitable *symbolic models*. Such models should provide a compact computer representation for control and data flows, and should be appropriate for mainstream verification techniques such as model checking and theorem proving. A number of symbolic models have been proposed, many of which based on the *guarded commands* (also known as *condition/action*) paradigm. In this paper, we draw attention to the limitations of this paradigm and propose a better model named NTIF (*New Technology Intermediate Form*), which is well-adapted to compiling high-level, concurrent languages (such as the recent E-LOTOS standard). Finally, we present two software tools developed for NTIF and report about the use of NTIF for modeling two embedded applications in smart cards.

1 Introduction

In computer-aided verification of concurrent systems, one usually distinguishes between:

- *High-level languages* used to describe the concurrent systems to be verified. These languages should be expressive and provide user-convenient features. Examples of such languages are Formal Description Techniques (such as the international standards LOTOS [14] and E-LOTOS [15]) as well as other languages for concurrent systems, e.g., PROMELA [13], etc.
- *Low-level models* on which verification is performed using dedicated algorithms. Examples of such models are Labeled Transition Systems, Kripke structures, Petri Nets and their corresponding marking graphs, Binary Decision Diagrams, etc.

* This work has been partly done in the framework of FORMALCARD (see http://www.inrialpes.fr/vasy/dyade/formalcard.html), a joint research project between INRIA and SCHLUMBERGER (formerly BULL/CP8)

D.A. Peled and M.Y. Vardi (Eds.): FORTE 2002, LNCS 2529, pp. 276–291, 2002.

In most cases, it is not feasible to describe a complex system manually using a low-level model. For this reason, low-level models are often derived from descriptions written in high-level languages using automatic translation. To do so, one needs to introduce *intermediate models*, which take place between high-level languages and low-level models. There are several reasons for this:

- Direct translation from high-level languages into low-level models is often complex due to language features intended for user convenience; instead, performing translation in several steps using intermediate forms is generally easier.
- Although low-level models are theoretically simple, the size of their computer representation can grow quickly due to, e.g., the *state explosion problem* occurring with systems that contain many asynchronous components and/or manipulate complex data structures. As this complexity may exceed the capabilities of verification algorithms, it is suitable to have intermediate models (simpler than high-level languages, but more concise and abstract than low-level models) on which various transformations, simplifications, and optimizations can be applied.

The need for intermediate models has been recognized for long. [19] presents an approach in which a high-level language (used to describe a set of sequential tasks that execute asynchronously and communicate using CSP-like primitives) is translated into a low-level model (a Kripke structure on which temporal logic formulas can be evaluated) using an intermediate model (a Petri net extended with variables, boolean conditions and assignments). [22] describes a first implementation of these ideas using a simpler intermediate model (a set of guarded commands instead of a Petri net). [10] translates a large subset of *full* LOTOS[1] into an intermediate model (a Petri net extended with data handling); this *network* model plays a central role in CAESAR, the LOTOS compiler of the CADP verification toolbox [8], and was later enriched by adding *reset* actions (in 1992) and *reactions* (in 1999). [16] builds upon this approach by replacing Petri nets by communicating state machines in order to support the dynamic creation/destruction of LOTOS processes. Other intermediate models are: *Input/Output Automata* [17], *Linear Process Operators* [2] (also known as *Linear Process Equations* [11]), *Symbolic Transition Systems* [12], IF version 1.0 [3] and 2.0 [4], etc.

A suitable intermediate model should provide a compact representation for both asynchronous concurrency and data handling, which are two major causes of state explosion. In this paper, we do not address concurrency issues, as well-known approaches (such as Petri nets and communicating state machines) already exist for modeling asynchronous concurrency concisely, without flattening the state space.

As regards data handling, a suitable intermediate model should be *symbolic* in the sense that it represents each variable as a first-order object instead of

[1] i.e., value-passing LOTOS, contrary to contemporary works [25,18] that only handle process algebras without data.

eliminating each variable by enumerating the set of all its possible values (the so-called *data expansion* used in non-symbolic model checkers). By avoiding data expansion, symbolic models allow to represent large systems concisely. In some cases, they even allow to represent infinite space systems finitely. Moreover, they can be analyzed, transformed, and optimized using data flow analysis techniques and later be verified using (non-symbolic or symbolic) model checking and theorem proving methods.

Although many symbolic models have been proposed in the literature, none of them seem appropriate for implementing the new Formal Description Technique E-LOTOS [15] recently standardized by ISO. Moreover, after a careful study of two applications embedded in smart cards (joint work with the SCHLUMBERGER company and the VERTECS team of INRIA Rennes), we have also reached the conclusion that the aforementioned symbolic models are not algorithmically optimal for symbolic analysis.

This paper addresses the issue of finding a general, symbolic model that could be used as an intermediate form for E-LOTOS (as well as other high-level languages, among which process algebras such as LOTOS) and would be suitable for efficient model checking and theorem proving. The paper is organized as follows. Section 2 points out some drawbacks of the existing symbolic models and lists requirements that a suitable intermediate form should satisfy. Section 3 defines a new symbolic model named NTIF (*New Technology Intermediate Form*) for describing communicating sequential processes with data. Section 4 presents two software tools developed for NTIF. Section 5 reports about the use of NTIF for modeling two smart card applications. Section 6 concludes the paper.

2 Rationale for NTIF

In this section, we discuss crucial criteria that a suitable intermediate model for E-LOTOS and LOTOS should satisfy. We focus on sequential processes expressed as state/transition machines with state variables. We assume that transitions are labeled with input/output communication events, which will serve for message-passing synchronisation when the machines are composed in parallel.

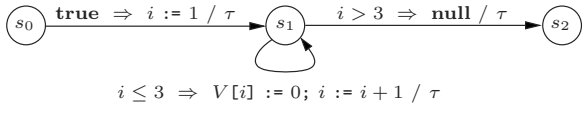
A suitable model should support conditions on input values. Most intermediate models are based on so-called *guarded commands*, also known as *condition/action* (see [20, chap. 4] for an example). Typically, in *condition/action* models, a transition from state s_1 to state s_2 has the form "$s_1 \xrightarrow{E \Rightarrow A \ / \ C} s_2$", where E is a condition that must be satisfied to fire the transition, A (called an *action*) is a sequence of variable assignments, and C is a *communication event*, being either (1) the internal event noted τ in process algebra theory, or (2) an input communication "G `?`V", where G is a gate (similar to CSP ports or LOTOS gates) and V is a variable used to store the value received on G, or (3) an output communication "G `!`E'", where E' is an expression, whose value is sent on G. Condition/action models may slightly differ with respect to the nature of actions (some models only allow a single variable assignment per transition) and

the precise order in which the condition, the action, and the communication are evaluated/executed. In the case of input communications, if expression E is evaluated before the input of V, then it can not be used to constrain the received value. This is not expressive enough for E-LOTOS and process algebra such as LOTOS, which allow to constrain the received values in input communications using conditions. For instance, in "G **?**V **:int where** $V \leq 3$" the transition only fires if the value received on gate G is less than 3.

A suitable model should allow conditions and actions to be intertwined. If a condition attached to a transition contains identical sub-expressions, then auxiliary variables should be introduced to avoid redundant computations. Thus, assignments to auxiliary variables should be allowed before evaluating the condition. For instance, if the condition is "$F(F'(X), F'(X))$", where F and F' are functions and X a state variable, it should be possible to introduce an assignment "X' := $F'(X)$" to an auxiliary variable X'. This assignment should be executed *before* evaluating the new condition "$F(X', X')$" that decides whether the transition is fireable (contrary to most condition/action models, in which actions are executed *after* evaluating conditions).

A suitable model should have a rich language of actions. The semantics of concurrent languages is either *small-step* or *big-step*. In a small-step semantics, each variable assignment creates a transition in the underlying graph model. In PROMELA for instance, the sequence of three assignments "X_1 := 0 ; X_2 := 0 ; X_3 := 0" creates three transitions by default. The user may decide to aggregate these transitions into a single one, by using explicitly a special operator "`dstep`". By contrast, E-LOTOS has a big-step semantics: variables are local to parallel processes (meaning that a variable assignment in one process is invisible from the other processes). There is no transition associated to assignments in the underlying graph model. Instead, transitions are created by communications only (either input, output, or internal). Thus, by default, the statement "X_1 := 0 ; X_2 := 0 ; X_3 := 0" does not create any transition unless the user decides to add explicit τ events between assignments.

In condition/action models "$s_1 \xrightarrow{E \Rightarrow A \ / \ C} s_2$", if the language of actions is simple (i.e., A consists in only one assignment), the semantics is small-step, since each assignment corresponds to one transition. Alternatively, a more complex language of actions (permitting several assignments in sequence) involves a mixture of small-step and big-step semantics. For instance, in IF version 1.0, it is possible to assign in the same transition say, 3 variables or even 3 array elements. However, since IF 1.0 does not support loops, it is not possible to implement correctly the action "**for** i **in** 1..3 **do** V[i] := 0 **end**", because τ-transitions must be introduced for testing and incrementing the loop index i. Concretely, such a statement must be unfolded in three transitions:



This example shows that the semantics clearly lacks consistency, since two equivalent statements ("**for** i **in** $1..3$ **do** V`[`i`]` `:=` 0 **end**" and "V`[1] := 0; `V`[2] := 0; `V`[3] := 0`") do not have the same semantic model. As long as we deal with purely sequential systems, τ-transitions can be eliminated by transitive closure. However, when systems are put in parallel, τ-transitions become meaningful in terms of branching time semantics. Generating extra τ-transitions does not preserve strong bisimulation and makes the state space grow excessively due to the presence of extra interleavings. Therefore, condition/action models are not appropriate for big-step semantics, for which a richer language of actions is needed.

A suitable model should not duplicate conditions. The translation of high-level control structures ("**if-then-else**", "**case**", "**while**" and "**for**" loops, etc.) into condition/action models requires conditions to be duplicated unnecessarily. For instance, the high-level statement "**if** E_1 **then** C_1 **elsif** E_2 **then** C_2 ... **elsif** E_n **then** C_n **else** C_{n+1} **end**" (where $C_1, \ldots, C_{n+1}$ are communications) is expanded into $n+1$ transitions:

$$s \xrightarrow{E_1 \Rightarrow \textbf{null} \ / \ C_1} s' \tag{1}$$

$$s \xrightarrow{\textbf{not}(E_1) \textbf{ and } E_2 \Rightarrow \textbf{null} \ / \ C_2} s' \tag{2}$$

...

$$s \xrightarrow{\textbf{not}(E_1) \textbf{ and } \ldots \textbf{ and not}(E_{n-1}) \textbf{ and } E_n \Rightarrow \textbf{null} \ / \ C_n} s' \tag{n}$$

$$s \xrightarrow{\textbf{not}(E_1) \textbf{ and } \ldots \textbf{ and not}(E_{n-1}) \textbf{ and not}(E_n) \Rightarrow \textbf{null} \ / \ C_{n+1}} s' \tag{n+1}$$

Starting with n conditions $E_1, \ldots E_n$ in the high-level description, the translation ends up with $n(n+1)/2$ conditions in the condition/action model.

Such an expansion makes the task of writing condition/action models "by hand" error-prone, and introduces a complexity overhead that penalizes the efficiency of automated analysis, either using model checking or theorem proving. Usually, a symbolic analysis tool used to analyse condition/action models (such as the OMEGA calculator used in the STG symbolic test generator [7]) has to decide which transitions can be fired from a given state, given constraints on the values of state variables. This is computationally expensive (and even undecidable in general) but can be optimized. For instance, knowing that a set of transitions are mutually exclusive, once one of them has proven fireable, it immediately follows that the others are not. Similarly, given that $n+1$ transitions are exhaustive (i.e., for all values of variables one of the transitions can be fired), once n of them have proven non-fireable then the last one can be fired. This information is usually present in high-level languages, but is lost after translation into condition/action models.

3 Definition of NTIF

As none of the models found in the literature meet all the criteria detailed in Section 2, we had to design a new model named NTIF.

3.1 Syntax

An NTIF automaton is a 10-tuple $\langle \mathcal{T}, \mathcal{C}, \mathcal{F}, \mathcal{V}, \mathcal{X}, E_0, \mathcal{G}, \mathcal{S}, s_0, act \rangle$ where:

- $\mathcal{T}$ is a set of *types* (noted $T, T', T_0, T_1, \ldots$), imported from libraries or defined using type expressions, which are left out of this paper.
- $\mathcal{C}$ is a set of *constructor symbols* (noted $C, C', C_0, C_1, \ldots$), each of which is characterized by the types of its arguments and result.
- $\mathcal{F}$ is a set of (non-constructor) *function symbols* (noted $F, F', F_0, F_1, \ldots$), each of which is also characterized by the types of its arguments and result. Functions are assumed to be total (exception handling is deferred to further work). The language of function definitions is left out of this paper.
- $\mathcal{V}$ is a finite set of global typed *variables* (noted $V, V', V_0, V_1, \ldots$).
- $\mathcal{X} \subseteq \mathcal{V}$ is a set of *formal parameters* (noted $X, X', X_0, X_1, \ldots$).
- E_0 is a boolean *expression*[2] that denotes an initial condition on the parameters $\mathcal{X}$.
- $\mathcal{G}$ is a set of *gates* (noted $G, G', G_0, G_1, \ldots$), including the special gate τ.
- $\mathcal{S}$ is a finite set of *states* (noted $s, s', s_0, s_1, \ldots$).
- $s_0 \in \mathcal{S}$ is the *initial state*.
- *act* is a function that associates to each state s an *action*[3]. Contrary to state/transition models, for a given s, $act(s)$ is unique and describes all the possible successor states of s, as well as effects on the state variables. Since $act(s)$ may lead to different successor states, *act* is called a *multi-branch transition relation*. Syntactically in an NTIF model, *act* is defined as a list "**from** s_1 $act(s_1) \ldots$ **from** s_n $act(s_n)$".

We now define *expressions* (noted $E, E', E_0, E_1, \ldots$), *patterns* ($P, P', P_0, \ldots$), *offers* ($O, O', O_0, \ldots$), and *actions* ($A, A', A_0, \ldots$), the abstract syntax of which is described in Figure 1. The following conventions are used for optional and repeated elements. Parts of the syntax enclosed in square brackets are optional. A list indexed from 1 to n (e.g., "$E_1, \ldots, E_n$") denotes a possibly empty sequence of symbols, whereas a list indexed from 0 to n always contains at least one element. Expressions, patterns, and offers are derived from E-LOTOS, whereas actions are specific to NTIF.

An expression is either a variable (rule R1), a constructor call (rule R2), or a function call (rule R3). A pattern is either an anonymous variable ("*wildcard*") of type T (rule R4), a variable (rule R5), a constructor call (rule R6), or a pattern P_0, the variables of which must satisfy a condition E (rule R7). An offer is either the emission of the value of an expression E (rule R8), or the receipt of a value that must match P using standard pattern-matching. Actions are described as follows:

- Rule R10 denotes the neutral element of sequential composition.
- Rule R11 denotes the vectorial assignment of variables $V_0, \ldots, V_n$ with the values of expressions $E_0, \ldots, E_n$.

[2] The syntax and semantics of expressions will be defined below.
[3] The syntax and semantics of actions will be defined below.

$$
\begin{array}{lll}
E ::= V & & \text{(R1)} \\
\quad \mid C(E_1, \ldots, E_n) & & \text{(R2)} \\
\quad \mid F(E_1, \ldots, E_n) & & \text{(R3)} \\
P ::= \textbf{any}\ T & & \text{(R4)} \\
\quad \mid V & & \text{(R5)} \\
\quad \mid C(P_1, \ldots, P_n) & & \text{(R6)} \\
\quad \mid P_0\ \textbf{where}\ E & & \text{(R7)} \\
O ::= \texttt{!}E & & \text{(R8)} \\
\quad \mid \texttt{?}P & & \text{(R9)} \\
A ::= \textbf{null} & & \text{(R10)} \\
\quad \mid V_0, \ldots, V_n\ \texttt{:=}\ E_0, \ldots, E_n & & \text{(R11)} \\
\quad \mid V_0, \ldots, V_n\ \texttt{:=}\ \textbf{any}\ T_0, \ldots, T_n\ [\textbf{where}\ E] & & \text{(R12)} \\
\quad \mid \textbf{reset}\ V_0, \ldots, V_n & & \text{(R13)} \\
\quad \mid G\ O_1\ \ldots\ O_n & & \text{(R14)} \\
\quad \mid \textbf{to}\ s & & \text{(R15)} \\
\quad \mid A_1\ \texttt{;}\ A_2 & & \text{(R16)} \\
\quad \mid \textbf{select}\ A_1\ \texttt{[]}\ \ldots\ \texttt{[]}\ A_n\ \textbf{end}\ [\textbf{select}] & & \text{(R17)} \\
\quad \mid \textbf{case}\ E\ \textbf{is}\ P_1 \rightarrow A_1\ \texttt{|}\ \ldots\ \texttt{|}\ P_n \rightarrow A_n\ \textbf{end}\ [\textbf{case}] & & \text{(R18)} \\
\quad \mid \textbf{while}\ E\ \textbf{do}\ A_0\ \textbf{end}\ [\textbf{while}] & & \text{(R19)}
\end{array}
$$

Fig. 1. Syntax of NTIF expressions, patterns, offers, and actions

- Rule R12 denotes the vectorial assignment of variables $V_0, \ldots, V_n$ with arbitrary values of respective types $T_0, \ldots, T_n$ such that the optional condition E (if present) is satisfied.
- In rule R13, the respective values of variables $V_0, \ldots, V_n$ are reset to the undefined value; in order to be re-used, these variables must be assigned new values.
- In rule R14, communication is performed on gate G and offers $O_1, \ldots, O_n$ (optional) model the data communications between processes.
- Rule R15 denotes a jump to state s.
- Rule R16 denotes the sequential composition of actions A_1 and A_2.
- Rule R17 denotes the arbitrary selection of one of the actions $A_1, \ldots, A_n$.
- Rule R18 denotes the selection of the first action A_i in $A_1, \ldots, A_n$ such that P_i matches the value of E.
- Rule R19 denotes a standard "**while**" loop that stops when E evaluates to false.

Useful standard shorthand notations are defined as follows:

if E **then** A_1 **else** A_2 **end** [**if**] = **case** E **is true** $\rightarrow A_1$ | **false** $\rightarrow A_2$ **end**

if E **then** A **end** [**if**] = **if** E **then** A **else null end**

for V **in** $E_1 \,..\, E_2$ **do** A **end** [**for**] = V := E_1; **while** $V \leq E_2$ **do** A; V := $V + 1$ **end**

stop = **select end** (or equivalently **case true is end**)

3.2 Informal Insight into NTIF Semantics

Before considering in more details the static and dynamic semantics of NTIF, we sum up briefly its intuitive semantics.

An NTIF automaton denotes a state/transition machine with state variables. Its semantics can be expressed in terms of a *Labeled Transition System* (LTS), the states of which are couples (s, ρ) consisting of a state s of the NTIF automaton and a *store* ρ assigning values to state variables. Each transition of the LTS is labeled by a communication event. Given a state (s, ρ), the outgoing transitions are calculated by executing the action $act(s)$ in the store ρ, which can produce either 0, 1, or several successor states (s', ρ') — hence the term *multi-branch* used to characterize the *act* function.

A state (s_1, ρ_1) of the LTS has no successor if the execution of $act(s_1)$ blocks before reaching a jump to a next state. We give four such examples for $act(s_1)$:

- "**null**": no jump specified
- "**while true do** X := $X + 1$ **end while; to** s_2": diverging loop execution
- "G ?V **where** $V < 10$ **and** $V > 20$**; to** s_2": impossible condition
- "**case** V **is** $1 \rightarrow$ **to** s_1 | $2 \rightarrow$ **to** s_2 **end**" with $\rho_1(V) = 3$: unexpected case

A state (s_1, ρ_1) has several successors if $act(s_1)$ is non-deterministic, either for control, such as in "**select** G_1**; to** s_1 ▯ G_2**; to** s_2 **end**", or for data, such as in "G ?V**; if** $V < 0$ **then to** s_1 **else to** s_2 **end**" or "V := **any int; if** $V < 0$ **then to** s_1 **else to** s_2 **end**".

The possibility of having multiple labels and output states in the same action is a distinctive flavor of NTIF, which differs from all aforementioned symbolic models, the transitions of which have exactly one label and one output state (or one fixed set of output places in the case of CAESAR's Petri nets [10]). To our knowledge, a similar feature can only be found in the IC (Intermediate Code) model used in the ESTEREL compiler [1]; however, IC is not a pure state/transition model and is targeted towards synchronous language implementation.

Note that NTIF generalizes the condition/action models, since the set of transitions "$s \xrightarrow{E_i \Rightarrow A_i \ / \ C_i} s_i$" $(0 \leq i \leq n)$ going out of a state s can be written "**from** s **select** C_0 **where** E_0**;** A_0**; to** s_0 ▯ ... ▯ C_n **where** E_n**;** A_n**; to** s_n **end**" in NTIF. It is also clear that NTIF meets the criteria given in Section 2:

- Conditions on input values are supported by means of conditional patterns, such as "V **where** $V \leq 3$".
- Conditions and actions can be intertwined arbitrarily, as in "X':= $F'(X)$**; if** $F(X', X')$ **then** G**; to** s' **end**".
- NTIF has a big-step semantics. For instance, the execution of action "**for** i **in** 1..3 **do** V[i] := 0 **end; to** s_2" produces a single transition in the underlying LTS.
- Duplication of conditions is avoided thanks to high-level "**if-then-else**", "**case**", "**while**", and "**for**" control structures.

3.3 Static Semantics

NTIF static semantics contains both standard analyses (not detailed here) such as variable binding and strong typing, and original ones based on control and data flow analysis.

The latter determine whether an NTIF automaton is accepted or rejected, based on its *static execution paths*, a compile-time over-approximation of its *dynamic execution paths* (i.e., walks in the program according to the dynamic semantics defined in Section 3.4). Of course, some programs may be rejected because some of their computed static execution paths do not satisfy a given property, although all dynamic execution paths will actually do. Nevertheless, this approach is practically acceptable because (1) it statically guarantees run-time properties of programs, and (2) code can always be rewritten (often gaining in clarity) in order to pass the static checks.

Variable initialization analysis (similar to HERMES, JAVA, and E-LOTOS) ensures that each variable is defined before being used (under the assumption that the formal parameters are initialized). This is useful to detect frequent programming mistakes at an early stage. Simple examples of NTIF code rejected by the analysis are "**reset** V_1 **;** V_2 **:=** V_1", or "**from** s V **:=** $V+1$ **; reset** V **; to** s". Also, "**reset** V **; if** E **then** V **:=** 1 **end; if not**(E) **then** V **:=** 0 **end;** G **!**V **; to** s" is rejected statically, although it would satisfy the variable initialization property at run-time. However, this code can be rewritten using an "**else**" clause instead of the second "**if**" and be accepted by the compiler. Variable initialization analysis extends smoothly to arrays by requiring global initialization, as in JAVA.

Unicity of communication analysis verifies that there is at most one communication on each execution path of an action $act(s)$. It is permitted to have different communications in the same action, such as in "**if** E **then** G_1 **else** G_2 **end**", because G_1 and G_2 cannot occur in the same execution path. However for instance, "G_1 **;** G_2", "**if** E **then** G_1 **end;** G_2", and "**while** E **do** G **end**" are forbidden.

This pragmatic restriction makes NTIF different from the language ESTELLE [5]. In ESTELLE, transitions are defined using structured PASCAL code and may contain several *outputs*. This causes both semantic and practical limitations, since for instance, the transition labels may become too large to be visualized as the number of outputs in the same transition may be unbounded e.g., in the case of a "**while**" loop containing outputs, which is incompatible with rendezvous synchronization. Moreover, ESTELLE transitions are not multi-branch as each transition has only one successor state.

Finally, *next state reachability* ensures that the execution of an action $act(s)$ can not be blocked between a communication and the next "**to**" action. For instance, both actions "G**;** V **:= any** T **where** E**; to** s" and "G**; if** E **then to** s **end**" are rejected because condition E could be false, whereas "G**;** V **:= any** T**; to** s" and "G**; if** E **then to** s_1 **else to** s_2 **end**" are accepted.

3.4 Dynamic Semantics

The dynamic semantics of NTIF associates an LTS to an NTIF automaton $\langle \mathcal{T}, \mathcal{C}, \mathcal{F}, \mathcal{V}, \mathcal{X}, E_0, \mathcal{G}, \mathcal{S}, s_0, act \rangle$, assuming that all parameters of $\mathcal{X}$ are instantiated properly. To this aim, the notions of *values*, *labels*, and *stores* are defined:

- *Values* (noted $v, v', v_0, v_1, \ldots$) are typed expressions built using constructors of $\mathcal{C}$ (*ground terms*). The set of values is noted $Val(\mathcal{T}, \mathcal{C})$.
- *Labels* (noted $\ell, \ell', \ell_0, \ell_1, \ldots$) are either tuples of the form $\langle G, v_1, \ldots, v_n \rangle$, or a special label ε corresponding to transitions without communication events. The set of labels is noted $Lab(\mathcal{T}, \mathcal{C}, \mathcal{G}) \subseteq (\mathcal{G} \times Val(\mathcal{T}, \mathcal{C})^*) \cup \{\varepsilon\}$. The partial operator $+$ is defined by $\ell + \varepsilon = \varepsilon + \ell = \ell$, and undefined if both its operands are different from ε.
- *Stores* (noted $\rho, \rho', \rho_0, \rho_1, \ldots$) are partial functions mapping variables to values. The set of stores is noted $Store(\mathcal{T}, \mathcal{C}, \mathcal{V}) \subseteq \mathcal{V} \rightarrow (Val(\mathcal{T}, \mathcal{C}) \cup \{undefined\})$. We note $[V_1 \mapsto v_1, \ldots, V_n \mapsto v_n]$ (where all V_i's are pairwise distinct) the store ρ such that $\rho(V_i) = v_i$ for all $i \in 1..n$, and undefined elsewhere. The *restriction* operator $\ominus$ and the *update* operator $\oslash$ are defined as follows:

$$(\rho \ominus \{V_1, \ldots, V_n\})(V) = \text{if } V \notin \{V_1, \ldots, V_n\} \text{ then } \rho(V) \text{ else } undefined$$
$$(\rho \oslash \rho')(V) = \text{if } \rho'(V) = undefined \text{ then } \rho(V) \text{ else } \rho'(V)$$

The semantics of expressions is given by a predicate $eval(E, \rho, v)$ that is true iff the evaluation of E in store ρ terminates and yields a value v. Expression evaluation is *side effect free* (it does not change the value of any variable) and *deterministic* (given E and ρ, there is at most one v such that $eval(E, \rho, v)$; there might be no such v if the evaluation of E diverges). The static semantics ensures that for each variable V used in E, $\rho(V)$ is defined and has the same type as V.

The semantics of patterns is given by a *pattern-matching* function $match(v, \rho, P)$ that returns either "**fail**" if value v does not match pattern P, or a new store ρ' corresponding to ρ in which the variables of P have been assigned by the matching sub-terms of v. The store ρ is used to evaluate guards E in patterns of the form "P_0 **where** E", which match a value v iff $match(v, \rho, P_0) = \rho'$ and $eval(E, \rho', \mathbf{true})$.

The semantics of offers is given by a function $accept(v, \rho, O)$, defined by:

$$accept(v, \rho, \texttt{!}E) = \text{if } eval(E, \rho, v) \text{ then } \rho \text{ else } \mathbf{fail}$$
$$accept(v, \rho, \texttt{?}P) = match(v, \rho, P)$$

The semantics of actions is given in SOS style by a transition relation noted "$[A], \rho \stackrel{\ell}{\Longrightarrow} s, \rho'$" (defined in Figure 2), where (1) ℓ is a label if A executes a communication labeled with ℓ, or ε otherwise, (2) s is a state if A terminates with a "**to** s" action, or "**none**" otherwise, and (3) ρ' is the store obtained after execution of A in ρ.

Execution of an NTIF automaton $\langle \mathcal{T}, \mathcal{C}, \mathcal{F}, \mathcal{V}, \mathcal{X}, E_0, \mathcal{G}, \mathcal{S}, s_0, act \rangle$ in a store ρ_0 assigning values to all parameters of $\mathcal{X}$ and satisfying $eval(E_0, \rho_0, \mathbf{true})$ (the parameters fulfill the initial condition) yields an LTS $\langle \Sigma, \mathcal{L}, \rightarrow, \sigma_0 \rangle$ such that:

$$\frac{}{[\textbf{null}], \rho \stackrel{\varepsilon}{\Longrightarrow} \textbf{none}, \rho} \qquad \frac{}{[\textbf{to}\ s], \rho \stackrel{\varepsilon}{\Longrightarrow} s, \rho}$$

$$\frac{eval(E_0, \rho, v_0)\ \wedge\ \ldots\ \wedge\ eval(E_n, \rho, v_n)}{[V_0, \ldots, V_n\ \textbf{:=}\ E_0, \ldots, E_n], \rho \stackrel{\varepsilon}{\Longrightarrow} \textbf{none}, \rho \oslash [V_0 \mapsto v_0, \ldots, V_n \mapsto v_n]}$$

$$\frac{(\forall i \in 0..n)\ v_i \in T_i\ \wedge\ eval(E, \rho \oslash [V_0 \mapsto v_0, \ldots, V_n \mapsto v_n], \textbf{true})}{[V_0, \ldots, V_n\ \textbf{:=}\ \textbf{any}\ T_0, \ldots, T_n\ \textbf{where}\ E], \rho \stackrel{\varepsilon}{\Longrightarrow} \textbf{none}, \rho \oslash [V_0 \mapsto v_0, \ldots, V_n \mapsto v_n]}$$

$$\frac{}{[\textbf{reset}\ V_0, \ldots, V_n], \rho \stackrel{\varepsilon}{\Longrightarrow} \textbf{none}, \rho \ominus \{V_0, \ldots, V_n\}}$$

$$\frac{\rho_1 = \rho\ \wedge\ (\forall i \in 1..n)\ accept(v_i, \rho_i, O_i) = \rho_{i+1} \neq \textbf{fail}}{[G\ O_1\ \ldots\ O_n], \rho \stackrel{\langle G, v_1, \ldots, v_n \rangle}{\Longrightarrow} \textbf{none}, \rho_{n+1}}$$

$$\frac{[A_1], \rho \stackrel{\ell_1}{\Longrightarrow} \textbf{none}, \rho'\ \wedge\ [A_2], \rho' \stackrel{\ell_2}{\Longrightarrow} s, \rho''\ \wedge\ \ell = \ell_1 + \ell_2}{[A_1\ \textbf{;}\ A_2], \rho \stackrel{\ell}{\Longrightarrow} s, \rho''} \qquad \frac{[A_1], \rho \stackrel{\ell}{\Longrightarrow} s, \rho'\ \wedge\ s \neq \textbf{none}}{[A_1\ \textbf{;}\ A_2], \rho \stackrel{\ell}{\Longrightarrow} s, \rho'}$$

$$\frac{[A_i], \rho \stackrel{\ell}{\Longrightarrow} s, \rho'}{[\textbf{select}\ A_1\ \square\ \ldots\ \square\ A_n\ \textbf{end}], \rho \stackrel{\ell}{\Longrightarrow} s, \rho'}$$

$$\frac{eval(E, \rho, v)\ \wedge\ match(v, \rho, P_i) = \rho_i \neq \textbf{fail}\ \wedge\ (\forall j < i)\ match(v, \rho, P_j) = \textbf{fail}\ \wedge\ [A_i], \rho_i \stackrel{\ell}{\Longrightarrow} s, \rho'}{[\textbf{case}\ E\ \textbf{is}\ P_1 \rightarrow A_1\ |\ \ldots\ |\ P_n \rightarrow A_n\ \textbf{end}], \rho \stackrel{\ell}{\Longrightarrow} s, \rho'}$$

$$\frac{eval(E, \rho, \textbf{true})\ \wedge\ [A_0\ \textbf{;}\ \textbf{while}\ E\ \textbf{do}\ A_0\ \textbf{end}], \rho \stackrel{\ell}{\Longrightarrow} s, \rho'}{[\textbf{while}\ E\ \textbf{do}\ A_0\ \textbf{end}], \rho \stackrel{\ell}{\Longrightarrow} s, \rho'} \qquad \frac{eval(E, \rho, \textbf{false})}{[\textbf{while}\ E\ \textbf{do}\ A_0\ \textbf{end}], \rho \stackrel{\varepsilon}{\Longrightarrow} \textbf{none}, \rho}$$

Fig. 2. Dynamic semantics of NTIF actions

- $\Sigma \subseteq \mathcal{S} \times Store(\mathcal{T}, \mathcal{C}, \mathcal{V})$ is the set of states.
- $\mathcal{L} \subseteq Lab(\mathcal{T}, \mathcal{C}, \mathcal{G})$ is the set of labels.
- $\sigma_0 = \langle s_0, \rho_0 \rangle$ is the initial state.
- $\rightarrow \subseteq \Sigma \times \mathcal{L} \times \Sigma$ is the transition relation defined by $\langle \langle s, \rho \rangle, \ell, \langle s', \rho' \rangle \rangle \in \rightarrow$ (noted $\langle s, \rho \rangle \stackrel{\ell}{\longrightarrow} \langle s', \rho' \rangle$) iff $(\exists \langle s_0, \rho_0 \rangle, \ldots, \langle s_n, \rho_n \rangle)\ s_0 = s\ \wedge\ \rho_0 = \rho\ \wedge\ (\forall i \in 0..n-1)\ [act(s_i)], \rho_i \stackrel{\varepsilon}{\Longrightarrow} s_{i+1}, \rho_{i+1}\ \wedge\ [act(s_n)], \rho_n \stackrel{\ell}{\Longrightarrow} s', \rho'$.

Each transition $(s_1, \rho_1) \stackrel{\ell}{\longrightarrow} (s_2, \rho_2)$ can be seen as one *macro-step* made of *micro-steps* defined by the relation $[A], \rho \stackrel{\ell}{\Longrightarrow} s', \rho'$.

4 Tools for NTIF

We have developed two software tools (NT2IF and NT2DOT) for handling NTIF models. These tools (6,600 lines of code) have been implemented using an original compiler construction technology [9,24] developed by the VASY team of INRIA.

NT2IF translates NTIF into IOSTS (*Input Output Symbolic Transition Systems*) [21], a lower level formalism used as input of the STG symbolic test gener-

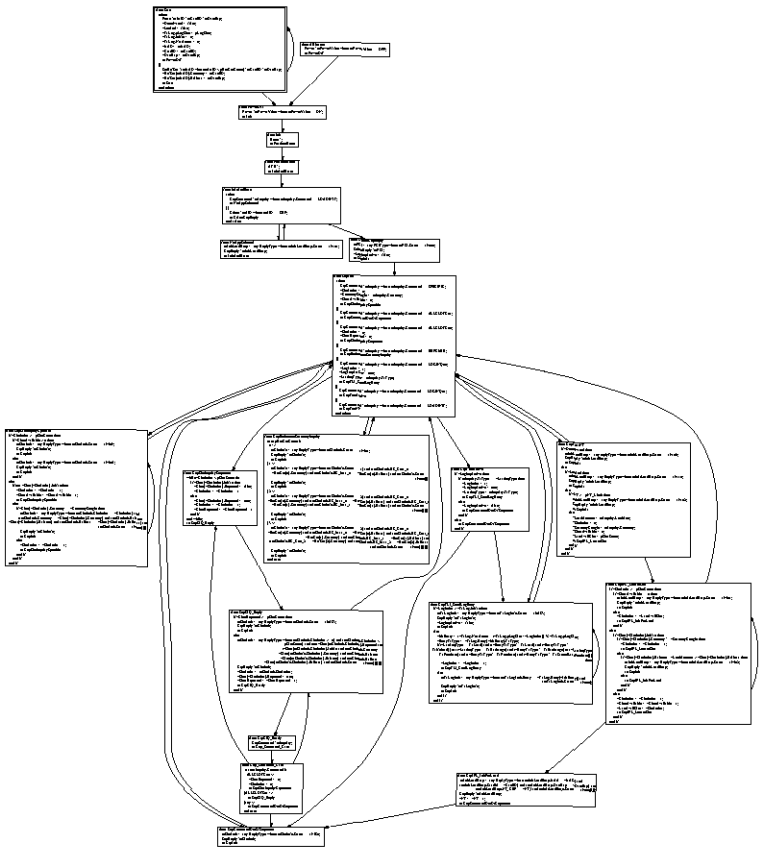


Fig. 3. Graphical representation of CEPS in NTIF (generated by NT2DOT).

ator [7] and based on the syntax of IF 1.0 [3]. NT2IF performs *symbolic unfolding*, i.e., it decomposes each NTIF action into (one or several) more elementary IF/IOSTS transitions, still preserving the semantics of the original NTIF model[4]. Thus, NT2IF allows NTIF to be used as a new, higher level input language for STG. This has proven appropriate in industrial case studies (see Section 5), as the lack of high-level control structures in IF/IOSTS constrains the user to introduce many intermediate states and transitions and to duplicate numerous conditions, which is a frequent source of mistakes.

NT2DOT provides a graphical representation of NTIF descriptions by translating them into the DOT format used by GRAPHVIZ, a graph visualization software developed by AT&T. Figure 3 shows an NTIF model viewed with NT2DOT. To each NTIF state s is associated a square box containing the code of $act(s)$ — one of these square boxes is detailed on Figure 4, providing an example of NTIF

[4] Modulo additional τ transitions as discussed in Section 2. Here, the τ transitions are harmless since parallel processes are not considered.

code. There is an arrow from box s_1 to box s_2 iff $act(s_1)$ contains an action "**to** s_2".

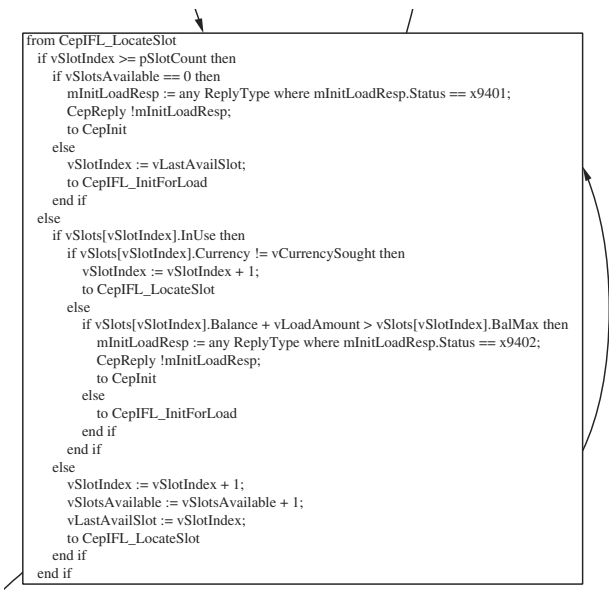


Fig. 4. Zoom on a state of Figure 3

5 Two Industrial Case Studies

Although NTIF is intended to be an intermediate model for E-LOTOS, it is sufficiently readable and structured to be written directly by humans. This use of NTIF is illustrated by two case studies performed in the framework of the FORMALCARD research project between INRIA and SCHLUMBERGER:

- We have modeled in NTIF a subset of CEPS (*Common Electronic Purse Specification*) [6], a multi-currency electronic purse application for smart cards. Starting from an IF/IOSTS model of CEPS developed by D. Clarke, we produced an equivalent NTIF model (represented on Figure 3), which was later translated back to IF/IOSTS automatically (using the NT2IF tool) to be processed by STG. The NT2IF translation algorithm was found clever enough, as the sizes of both IF/IOSTS models (the one written manually and the one generated automatically) are very close.
- F.X. Ponscarme used NTIF to model the administrative commands of a smart card operating system for 3GPP (*3rd Generation Partnership Project*) mobile telephony. His NTIF model was then translated into IF/IOSTS using NT2IF and processed by STG.

Both case-studies demonstrated that using NTIF instead of IF/IOSTS leads to *smaller, clearer, and safer* models. These findings are justified in the following paragraphs.

Figure 5 compares the sizes of the NTIF models w.r.t. the IF/IOSTS models generated by NT2IF. These sizes are measured in lines of code (type definitions excluded), states, and transitions. The columns labeled "% ↓" indicate the gain provided by NTIF given by the formula "(IF size - NTIF size) ×100 / IF size". The branching factor line gives the average number of successor states in each transition (thus, highlighting NTIF's multi-branch structure).

Besides size reduction, the existence of high-level control structures in NTIF leads to clearer models than with condition/action languages such as IF/IOSTS. In the case of CEPS, this is obvious from an immediate comparison of two graphical representations, the one generated by NT2DOT for the NTIF model (Figure 3) and the one generated by STG for the IF/IOSTS model (Figure 6), the latter being hardly readable even after enlargement.

The high-level control structures of NTIF are also safer than conditions/actions found in IF/IOSTS, which are error-prone because of the duplication of conditions (see Section 2). In particular, the modeling of nested "**if**" and "**case**" statements using conditions/actions is often erroneous, as some conditions are not always mutually exclusive as they would be expected to, or do not cover all possible cases. For instance, the reengineering work done for producing an NTIF model of CEPS from an IF/IOSTS one revealed more than 10 such bugs in the IF/IOSTS model. The conclusion was that this source of mistakes could be eliminated by using NTIF directly together with the NT2IF translator.

	Electronic Purse			Smart Card OS 3G		
	IF	NTIF	% ↓	IF	NTIF	% ↓
Number of lines	598	418	30 %	697	498	28 %
Number of states	31	21	32 %	34	20	41 %
Number of transitions	63	23	63 %	78	22	71 %
Branching factor	1	2,21	—	1	2,77	—

Fig. 5. Comparisons of NTIF and IF description sizes.

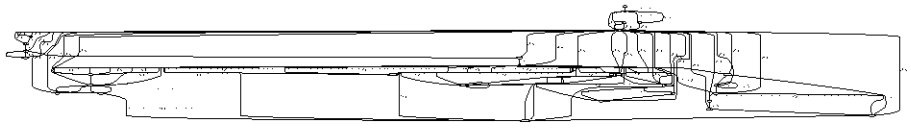


Fig. 6. Graphical representation of CEPS in IF (generated by STG).

6 Conclusion

In this paper, we have revisited the guarded commands model, which is popular both in concurrency theory textbooks and software tools. We have shown that this model has both theoretical limitations (conditions/actions are inappropriate for big-step semantics) and practical drawbacks (it increases the complexity of models to be analyzed).

To address these issues, we have proposed NTIF, a general, symbolic model for communicating sequential processes with data. Three distinctive features of NTIF are: its language of actions (which allows conditions and actions to be combined freely), its multi-branch structure (which enables conditions to be factorized), and its built-in support for pattern-matching. We have used NTIF in two industrial case-studies (smart card applications), which demonstrated the effective benefits of NTIF in practice.

We believe that, for efficiency reasons, future tools — including compilers, program transformers and optimizers, static analyzers, enumerative and symbolic model checkers, and theorem provers — will be based on NTIF-like models rather than classical condition/action models, which are clearly suboptimal. NTIF is expressive and general enough to be used as a target for the translation of several existing models [2,12,16,3,17,10]. Details about expressiveness will appear in a future paper.

Further work will consist in extending NTIF to support *concurrent processes* (which can be done easily using the parallel composition operators of process algebras such as LOTOS and E-LOTOS), *exceptions* arising from the computation of data, and *quantitative time*. We intend to reuse some of the proposals made in [23] for addressing these issues. With these extensions, NTIF will be used as a common intermediate model for both E-LOTOS and LOTOS.

Acknowledgements. The authors are grateful to the anonymous referees for their advised comments. They are also grateful to Kafia Zemirli (SCHLUMBERGER), for her constant support of the FORMALCARD project, as well as people from the VERTECS team of INRIA Rennes: Duncan Clarke and François-Xavier Ponscarme for their collaboration on case studies; Thierry Jéron, Vlad Rusu, and Bertrand Jeannet for their feedback about NTIF. They finally thank their colleagues Radu Mateescu and Frédéric Tronel for their valuable comments about the present paper.

References

1. G. Berry, G. Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science Of Computer Programming*, 19(2), 1992.
2. M. Bezem, J. Groote. Invariants in Process Algebra with Data. Proc. *CONCUR'94*, *LNCS* 836.
3. M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J.-P. Krimm, L. Mounier. IF: An Intermediate Representation and Validation Environment for Timed Asynchronous Systems. Proc. *FM'99*, *LNCS* 1708.

4. M. Bozga, S. Graf, L. Mounier. IF-2.0: A Validation Environment for Component-Based Real-Time Systems Proc. CAV'2002, *LNCS* 2404.
5. S. Budkowski, P. Dembinski. An Introduction to Estelle: A Specification Language for Distributed Systems. *Computer Networks and ISDN Systems*, 14(1), 1988.
6. CEPSCO. Common Electronic Purse Specification – Technical Specification version 2.3, 1999. `http://www.cepsco.com/`.
7. D. Clarke, T. Jéron, V. Rusu, E. Zinovieva. STG: A Symbolic Test Generation Tool. Proc. TACAS'2002, *LNCS* 2280.
8. H. Garavel, F. Lang, R. Mateescu. An Overview of CADP 2001. Technical Report RT 254, INRIA, 2001.
9. H. Garavel, F. Lang, R. Mateescu. Compiler Construction using LOTOS NT. Proc. *Compiler Construction 2002*, *LNCS* 2304.
10. H. Garavel, J. Sifakis. Compilation and Verification of LOTOS Specifications. Proc. *PSTV'90*. North-Holland.
11. J. Groote, M. Reniers. *Algebraic Process Verification*. Proc. *Handbook of Process Algebra*, chapter 17. North Holland, 2001.
12. M. Hennessy, M. Lin. Symbolic Bisimulations. *Theoretical Computer Science*, 138, 1995.
13. G. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), 1997.
14. ISO/IEC. LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, 1989.
15. ISO/IEC. Enhancements to LOTOS (E-LOTOS). International Standard 15437, 2001.
16. G. Karjoth. Implementing LOTOS Specifications by Communicating State Machines. Proc. *CONCUR'92*, *LNCS* 630.
17. N. Lynch, M. Tuttle. An Introduction to I/O automata. *CWI-Quarterly*, 2(3), 1989.
18. E.-R. Olderog. *Nets, Terms and Formulas*, *Cambridge Tracts in Theoretical Computer Science* 23. Cambridge University Press, 1991.
19. J.-P. Queille. *Le systéme CESAR : description, spécification et analyse des applications réparties*. Université Scientifique et Médicale de Grenoble (Grenoble, France) 1982.
20. W. de Roever, F. de Boer, U. Hanneman, J. Hooman, Y. Lakhnech, M. Poel, J. Zwiers. *Concurrency Verification – Introduction to Compositional and Noncompositional Methods*, *Cambridge Tracts in Theoretical Computer Science* 54. 2001.
21. V. Rusu, L. du Bousquet, T. Jéron. An Approach to Symbolic Test Generation. Proc. *IFM'00*, *LNCS* 1945.
22. J.-P. Schwartz. *QUASAR, une réalisation du système CESAR: description, spécification et analyse des applications réparties.* Thèse de Doctorat, Institut National Polytechnique de Grenoble (France), 1983.
23. M. Sighireanu. *Contribution à la définition et à l'implémentation du langage "Extended LOTOS"*. Thèse de Doctorat, Université Joseph Fourier (Grenoble, France), 1999.
24. M. Sighireanu. *LOTOS NT User's Manual (Version 2.1)*. INRIA projet VASY. `ftp://ftp.inrialpes.fr/pub/vasy/traian/manual.ps.Z`, 2000.
25. D. Taubner. *Finite Representations of CCS and TCSP Programs by Automata and Petri Nets*, *LNCS* 369. 1989.