

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

Hubert Garavel, Frédéric Lang

▶ **To cite this version:**

Hubert Garavel, Frédéric Lang. NTIF: A General Symbolic Model for Communicating Sequential Processes with Data. [Research Report] RR-4666, INRIA. 2002. inria-00071919

HAL Id: inria-00071919
https://inria.hal.science/inria-00071919

Submitted on 23 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

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

Hubert Garavel — Frédéric Lang

N° 4666

Décembre 2002

THÈME 1



ISSN 0249-6399



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

Hubert Garavel* , Frédéric Lang†

Thème 1 — Réseaux et systèmes
Projet VASY

Rapport de recherche n° 4666 — Décembre 2002 — 30 pages

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 report, 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.

Key-words: compilation, concurrent languages, concurrent systems, condition/action, guarded commands, modeling, symbolic model, verification

A short version of this report is also available as "*NTIF: a General Symbolic Model for Communicating Sequential Processes with Data*", in D. A. Peled and M. Y. Vardi, editors, Proceedings of the 22nd IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems FORTE'2002 (Houston, Texas), November 11-14, 2002.

* Hubert.Garavel@inria.fr
† Frederic.Lang@inria.fr

Unit´e de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 Montbonnot-St-Martin (France)
T´el´ephone : +33 4 76 61 52 00 — T´el´ecopie +33 4 76 61 52 52

NTIF : un modèle symbolique général pour les processus séquentiels communicants avec données

Résumé : Un problème central de la vérification assistée par ordinateur de systèmes séquentiels concurrents contenant des données est de trouver des *modèles symboliques* adaptés. De tels modèles devraient permettre une représentation compacte du flôt de contrôle et du flôt de données, et être appropriées pour les techniques de vérification telles que la vérification énumérative et la preuve de théorèmes. De nombreux modèles symboliques ont été proposés, la plupart étant basés sur le paradigme des *commandes gardées* (également appelé *condition/action*). Dans ce rapport, nous montrons les limites de ce paradigme et proposons un meilleur modèle nommé NTIF (*New Technology Intermediate Form*), qui est bien adapté à la compilation de langages concurrents de haut niveau (tel que la récente norme E-LOTOS). Enfin, nous présentons deux logiciels développés pour NTIF et nous faisons le bilan de l'utilisation de NTIF pour modéliser deux applications embarquées sur carte à puce.

Mots-clés : commandes gardées, compilation, condition/action, langages concurrents, modèle symbolique, modélisation, systèmes concurrents, vérification

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 [15] and E-LOTOS [16]) as well as other languages for concurrent systems, e.g., PROMELA [14], 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.

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. [20] 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). [23] describes a first implementation of these ideas using a simpler intermediate model (a set of guarded commands instead of a Petri net). [11] 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 CÆSAR, the LOTOS compiler of the CADP verification toolbox [9], and was later enriched by adding *reset* actions (in 1992) and *reactions* (in 1999). [17] builds upon

[1] i.e., value-passing LOTOS, contrary to contemporary works [26, 19] that only handle process algebras without data.

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* [18], *Linear Process Operators* [2] (also known as *Linear Process Equations* [12]), *Symbolic Transition Systems* [13], 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 report, 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 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 [16] 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 report 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 report 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 report.

2 Rationale for NTIF

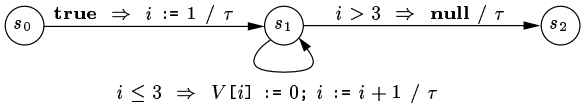
In this section, we discuss crucial criteria that a suitable intermediate model for E-LOTOS and LOTOS should satisfy. We focus on sequential processes expressed as state/transition machines with state variables. We assume that transitions are labeled with input/output communication events, which will 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 [21, 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 or equal to 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:

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

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

...

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

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

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 $T, T', T_0, T_1, \dots$), imported from libraries or defined using type expressions, which are left out of this report.
- $\mathcal{C}$ is a set of *constructor symbols* (noted $C, C', C_0, C_1, \dots$), each of which is characterized by the types of its arguments and result.
- $\mathcal{F}$ is a set of (non-constructor) *function symbols* (noted $F, F', F_0, F_1, \dots$), 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 report.
- $\mathcal{V}$ is a finite set of global typed *variables* (noted $V, V', V_0, V_1, \dots$).
- $\mathcal{X} \subseteq \mathcal{V}$ is a set of *formal parameters* (noted $X, X', X_0, X_1, \dots$).
- 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 $G, G', G_0, G_1, \dots$), including the special gate τ.
- $\mathcal{S}$ is a finite set of *states* (noted $s, s', s_0, s_1, \dots$).
- $s_0 \in \mathcal{S}$ 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

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

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.

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

Figure 1: Syntax of NTIF expressions, patterns, offers, and actions

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 $E_0, \ldots, E_n$.
- 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 CÆSAR's Petri nets [11]). 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 checking rules and algorithms mentioned in this section are given in the appendix.

Binding analysis aims at verifying that each occurrence of a variable is non-ambiguously bound to its definition. Variables occurring in an expression E, a pattern P, or an offer O are partitioned into sets of respectively *used* and *defined* variables: a variable is *defined* if its value may be changed by evaluation; it is *used* otherwise. For instance, all variables occurring in an expression are used variables because expression evaluation is free of side effect, whereas in the pattern "$C(V_1)$ **where** $V_1 \leq V_2$", V_1 is defined and V_2 is used. Examples of binding rules are the following:

- In an NTIF automaton of initial condition E_0 and parameters $\mathcal{X}$, the variables of E_0 are elements of $\mathcal{X}$ i.e., E_0 only constrains parameters.
- In rule (R6), every variable is defined at most once in the same pattern and a variable should not be used before being defined, taking into account that $P_1, \ldots, P_n$ are evaluated from left-to-right. For instance, "$C(V, V)$" and "$C(V_1$ **where** $V_1 = V_2, V_2)$" are rejected, but "$C(V_1, V_2$ **where** $V_1 = V_2)$" and "$C(V_1, V_2)$ **where** $V_1 = V_2$" are accepted. A similar rule exists for offers —also evaluated from left to right— in rule (R14).

Typing analysis aims at verifying that no typing conflict may arise at run-time. It relies on a standard —not detailed here— type-checking procedure for patterns and expressions, rejecting any automaton containing untyped objects. Additional examples of typing rules are the following:

- The initial condition E_0 of an NTIF automaton has type **bool**.
- In rules (R12), E has type **bool** and each V_i has type T_i.
- In rule (R18), each P_i has the same type as E.

We do not consider here the existence of gate types, a way of typing the interfaces between parallel processes [8]. The only constraint in rule (14) is $n = 0$ if $G = \tau$.

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:

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

The semantics of expressions is given by a predicate $\mathit{eval}(E, \rho, v)$ 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 $\mathit{eval}(E, \rho, v)$; 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 $\mathit{match}(v, \rho, P)$ 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 $\mathit{match}(v, \rho, P_0) = \rho'$ and $\mathit{eval}(E, \rho', \mathbf{true})$.

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

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

The semantics of actions is given in SOS style by a transition relation noted "$[A], \rho \xrightarrow{\ell} s, \rho'$" (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 $\langle \mathcal{T}, \mathcal{C}, \mathcal{F}, \mathcal{V}, \mathcal{X}, E_0, \mathcal{G}, \mathcal{S}, s_0, \mathit{act} \rangle$ in a store ρ_0 assigning values to all parameters of $\mathcal{X}$ and satisfying $\mathit{eval}(E_0, \rho_0, \mathbf{true})$ (the parameters fulfill the initial condition) yields an LTS $\langle \Sigma, \mathcal{L}, \rightarrow, \sigma_0 \rangle$ such that:

- $\Sigma \subseteq \mathcal{S} \times \mathit{Store}(\mathcal{T}, \mathcal{C}, \mathcal{V})$ is the set of states.
- $\mathcal{L} \subseteq \mathit{Lab}(\mathcal{T}, \mathcal{C}, \mathcal{G})$ 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 $\langle s, \rho \rangle \xrightarrow{\ell} \langle s', \rho' \rangle$) iff $(\exists \langle s_0, \rho_0 \rangle, \ldots, \langle s_n, \rho_n \rangle)\ s_0 = s \ \wedge \ \rho_0 = \rho \ \wedge \ (\forall i \in 0..n-1)\ [\mathit{act}(s_i)], \rho_i \xRightarrow{\varepsilon} s_{i+1}, \rho_{i+1} \ \wedge \ [\mathit{act}(s_n)], \rho_n \xRightarrow{\ell} s', \rho'$.

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

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

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

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

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

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

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

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

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

$$\frac{eval(E, \rho, \mathbf{true})\ \wedge\ [A_0\,\mathtt{;}\ \mathbf{while}\ E\ \mathbf{do}\ A_0\ \mathbf{end}], \rho \stackrel{\ell}{\Longrightarrow} s, \rho'}{[\mathbf{while}\ E\ \mathbf{do}\ A_0\ \mathbf{end}], \rho \stackrel{\ell}{\Longrightarrow} s, \rho'}$$

$$\frac{eval(E, \rho, \mathbf{false})}{[\mathbf{while}\ E\ \mathbf{do}\ A_0\ \mathbf{end}], \rho \stackrel{\varepsilon}{\Longrightarrow} \mathbf{none}, \rho}$$

Figure 2: Dynamic semantics of NTIF actions

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 [10, 25] developed by the VASY team of INRIA.

NT2IF translates NTIF into IOSTS (*Input Output Symbolic Transition Systems*) [22], a lower level formalism used as input of the STG symbolic test generator [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 graphics produced 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 code. There is an arrow from box s_1 to box s_2 iff $act(s_1)$ contains an action "**to** s_2".

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.

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

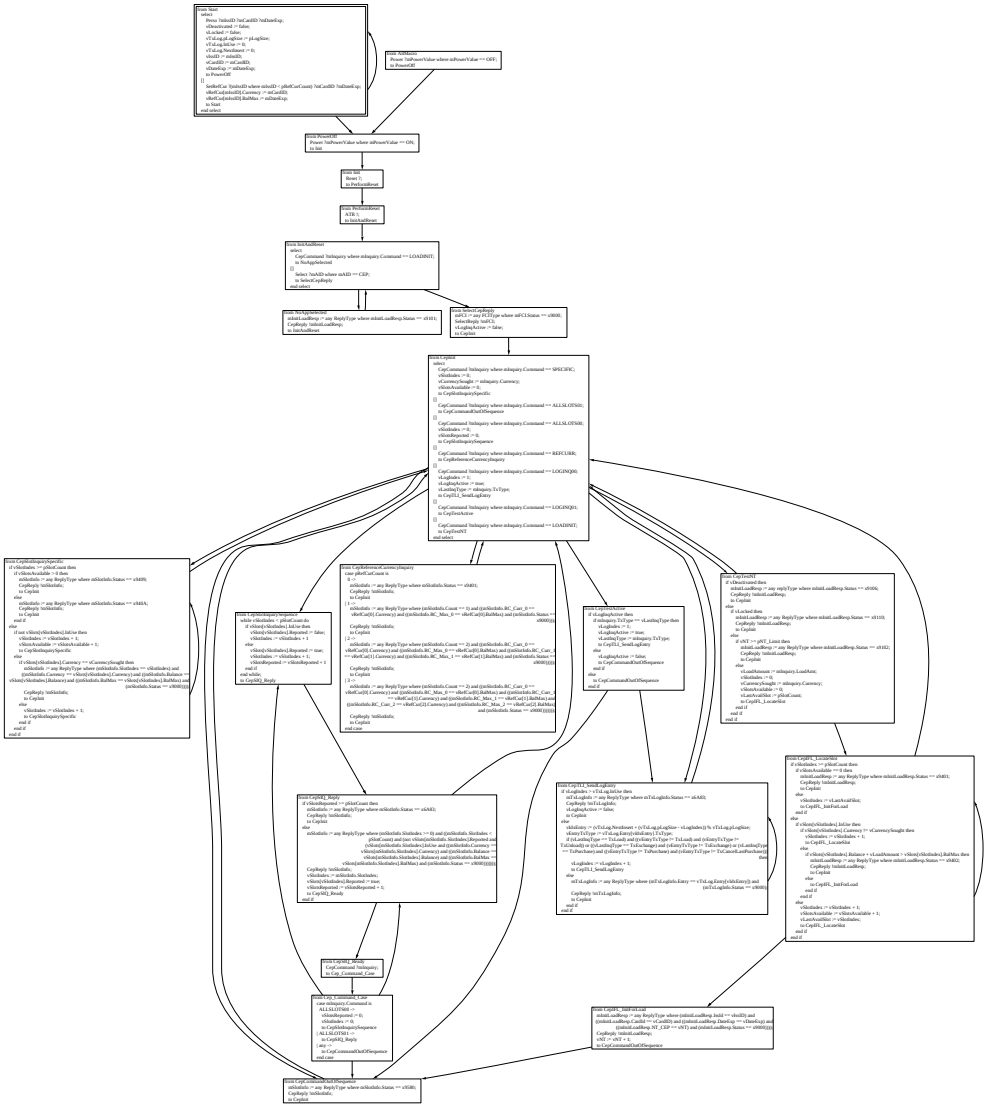


Figure 3: Graphical representation of CEPS in NTIF (generated by NT2DOT).

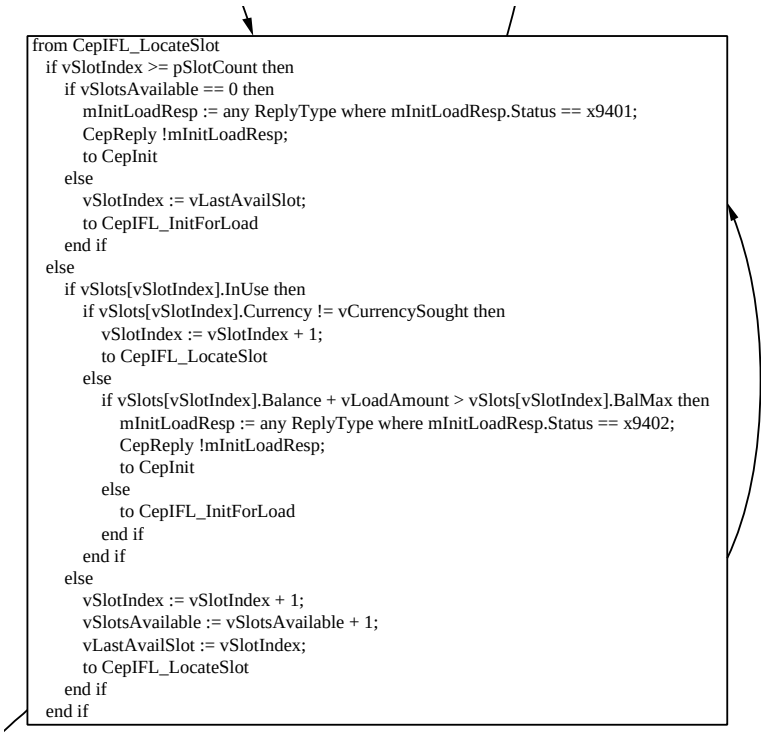


Figure 4: Zoom on a state of Figure 3

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	—

Figure 5: Comparisons of NTIF and IF description sizes.

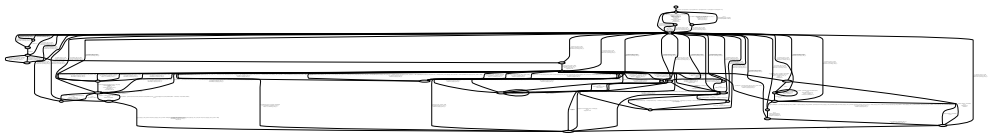


Figure 6: Graphical representation of CEPS in IF (generated by STG).

6 Conclusion

In this report, 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, 13, 17, 3, 18, 11]. 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 [24] 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 report.

References

[1] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, 19(2), 1992.

[2] M.A. Bezem and J.F. Groote. Invariants in process algebra with data. In *CONCUR'94*, volume 836 of *LNCS*, 1994.

[3] M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J.-P. Krimm, and L. Mounier. IF: An Intermediate Representation and Validation Environment for Timed Asynchronous Systems. In *FM'99*, volume 1708 of *LNCS*, 1999.

[4] Marius Bozga, Susanne Graf, and Laurent Mounier. IF-2.0: A Validation Environment for Component-Based Real-Time Systems. In Kim G. Larsen and Ed Brinksma, editors, *Proceedings of the Conference on Computer-Aided Verification CAV'2002 (Copenhagen, Denmark)*, volume 2404 of *Lecture Notes in Computer Science*. Springer Verlag, July 2002.

[5] S. Budkowski and 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. Available at `http://www.cepsco.com/`.

[7] Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. STG: A Symbolic Test Generation Tool. In Joost-Pieter Katoen and Perdita Stevens, editors, *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2002 (Grenoble, France)*, volume 2280 of *Lecture Notes in Computer Science*. Springer Verlag, April 2002.

[8] H. Garavel. On the Introduction of Gate Typing in E-LOTOS. In *PSTV'95*. Chapman & Hall, 1995.

[9] H. Garavel, F. Lang, and R. Mateescu. An Overview of CADP 2001. Rapport technique RT 254, INRIA, 2001.

[10] H. Garavel, F. Lang, and R. Mateescu. Compiler Construction using LOTOS NT. In *Compiler Construction 2002*, volume 2304 of *LNCS*, 2002.

[11] H. Garavel and J. Sifakis. Compilation and Verification of LOTOS Specifications. In *PSTV'90*. North-Holland, 1990.

[12] J.F. Groote and M.A. Reniers. *Algebraic Process Verification*. In *Handbook of Process Algebra*, chapter 17. North Holland, 2001.

[13] M. Hennessy and M. Lin. Symbolic Bisimulations. *Theoretical Computer Science*, 138, 1995.

[14] G. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), 1997.

[15] ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, ISO, 1989.

[16] ISO/IEC. Enhancements to LOTOS (E-LOTOS). International Standard 15437, ISO, 2001.

[17] G. Karjoth. Implementing LOTOS Specifications by Communicating State Machines. In *CONCUR'92*, volume 630 of *LNCS*, 1992.

[18] N. Lynch and M. Tuttle. An Introduction to Input/Output automata. *CWI-Quarterly*, 2(3), 1989.

[19] E.-R. Olderog. *Nets, Terms and Formulas*, volume 23 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1991.

[20] Jean-Pierre Queille. *Le systéme CESAR : description, spécification et analyse des applications réparties*. Université Scientifique et Médicale de Grenoble, Grenoble, 1982.

[21] W.P. de Roever, F. de Boer, U. Hanneman, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification – Introduction to Compositional and Noncompositional Methods*, volume 54 of *Cambridge Tracts in Theoretical Computer Science*. 2001.

[22] V. Rusu, L. du Bousquet, and T. Jéron. An Approach to Symbolic Test Generation. In *IFM'00*, volume 1945 of *LNCS*, 2000.

[23] Jean-Philippe Schwartz. *QUASAR, une réalisation du système CESAR: description, spécification et analyse des applications réparties.* Thèse de Doctorat, Institut National Polytechnique de Grenoble (France), November 1983.

[24] Mihaela Sighireanu. *Contribution à la définition et à l'implémentation du langage "Extended LOTOS"*. Thèse de Doctorat, Université Joseph Fourier (Grenoble), 1999.

[25] Mihaela Sighireanu. LOTOS NT User's Manual (Version 2.1). INRIA projet VASY. `ftp://ftp.inrialpes.fr/pub/vasy/traian/manual.ps.Z`, November 2000.

[26] D. Taubner. *Finite Representations of CCS and TCSP Programs by Automata and Petri Nets*, volume 369 of *LNCS*. 1989.

A Static Semantics

A.1 Binding

In the tables below, we define the following:

- the sets $use(E), use(P)$, and $use(O)$ of variables used in, respectively, an expression E, a pattern P, and an offer O
- the sets $def(E), def(P)$, and $def(O)$ of variables defined in, respectively, E, P, and O

Intuitively, a variable is used in a term if it must be assigned a value in order to enable the term evaluation. It is defined if it is assigned (or re-assigned) a value during term evaluation.

E	$use(E)$	$def(E)$
V	$\{V\}$	$\emptyset$
$C(E_1, \ldots, E_n)$	$\bigcup_{i=1}^{n} use(E_i)$	$\emptyset$
$F(E_1, \ldots, E_n)$	$\bigcup_{i=1}^{n} use(E_i)$	$\emptyset$

P	$use(P)$	$def(P)$
V	$\emptyset$	$\{V\}$
$C(P_1, \ldots, P_n)$	$\bigcup_{i=1}^{n} use(P_i)$	$\bigcup_{i=1}^{n} def(P_i)$
P_0 **where** E	$(use(E) \setminus def(P_0)) \cup use(P_0)$	$def(P_0)$
any T	$\emptyset$	$\emptyset$

O	$use(O)$	$def(O)$
$!E$	$use(E)$	$def(E)$
$?P$	$use(P)$	$def(P)$

Note that the sets $use(P)$ and $def(P)$ (respectively $use(O)$ and $def(O)$) are not necessarily disjoint, and that a variable may have several occurrences in which it is defined. This situation may turn the semantics of patterns counter-intuitive, and cause lots of programming errors. In order to palliate this, the notion of *well-binding* is defined.

A pattern P is *well-bound* if it satisfies the following rules:

- P is a variable or it has the form "**any** T".
- P has the form "$C(P_1\,,\ \ldots\,,\ P_n)$" and
  - each P_i $(i \in 1..n)$ is well-bound, and
  - for all $i \neq j, def(P_i) \cap def(P_j) = \emptyset$ (i.e., every variable is defined at most once in the same pattern), and
  - for all $i < j, def(P_j) \cap use(P_i) = \emptyset$ (i.e., a variable should not be used before being defined, patterns being evaluated from left to right).

- P has the form "P_0 **where** E" and P_0 is well-bound.

An action A is well-bound if it satisfies the following rules:

- A is a **null** or **to** statement.
- A has one of the forms "**reset** V_0 **,** $\ldots$ **,** V_n", "V_0 **,** $\ldots$ **,** V_n **:=** E_0 **,** $\ldots$ **,** E_n" or "V_0 **,** $\ldots$ **,** V_n **:= any** T_0 **,** $\ldots$ **,** T_n **where** E", and all V_i's are pairwise distinct (i.e., each variable may be assigned at most one value simultaneously).
- A has the form "$G\ O_1\ \ldots\ O_n$" and
  - for all O_i of the form "?P_i", P_i is well-bound, and
  - for all $i \neq j$, $def(O_i) \cap def(O_j) = \emptyset$, and
  - for all $i < j$, $def(O_j) \cap use(O_i) = \emptyset$ (similarly to patterns, offers are evaluated from left to right).
- A has the form "A_1 **;** A_2" and both A_1 and A_2 are well-bound.
- A has the form "**select** A_1 **[]** $\ldots$ **[]** A_n **end**" and each A_i ($i \in 1..n$) is well-bound.
- A has the form "**case** E **is** $P_1 \rightarrow A_1$ **|** $\ldots$ **|** $P_n \rightarrow A_n$ **end**", and each P_i, A_i ($i \in 1..n$) is well-bound.
- A has the form "**while** E **do** A_0 **end**" and A_0 is well-bound.

Finally, an NTIF automaton $\langle \mathcal{T}, \mathcal{C}, \mathcal{F}, \mathcal{V}, \mathcal{X}, E_0, \mathcal{G}, \mathcal{S}, s_0, act \rangle$ is well-bound iff $use(E_0) \subseteq \mathcal{X}$ (i.e., E_0 is a condition on the parameters) and for all $s \in S$, $act(s)$ is well-bound.

Note that well-binding does not ensure that variables are initialized before used. This is verified using a more complex algorithm defined in Section A.3.

A.2 Typing

Recall that in the definition of an NTIF automaton, a unique type is associated to each variable V, and prototypes of the form $T_1 \times \cdots \times T_n \rightarrow T$ are associated to constructor and function symbols.

Given these asumptions, the type of an expression E (respectively a pattern P, a value v), written $type(E)$ (respectively $type(P), type(v)$), is defined as follows:

- If T is the type associated to variable V, we write $type(V) = T$.
- $type($**any** $T) = T$.
- If the constructor symbol C (respectively the function symbol F) has prototype $T_1 \times \cdots \times T_n \rightarrow T$, and if $E_1, \ldots, E_n$ are expressions of respective types $T_1, \ldots, T_n$, then $type(C(E_1, \ldots, E_n)) = T$ (respectively $type(F(E_1, \ldots, E_n)) = T$).

- If $type(E) = \textbf{bool}$ then $type(P_0 \textbf{ where } E) = type(P_0)$ (it is undefined otherwise).

An action A is well-typed if it satisfies one of the following rules:

- A is a **null**, **reset**, or **to** statement.
- A has the form "V_0 `,` $\ldots$ `,` V_n `:=` E_0 `,` $\ldots$ `,` E_n", and for all $i \in 0..n$, $type(E_i) = type(V_i)$.
- A has the form "V_0 `,` $\ldots$ `,` V_n `:=` **any** T_0 `,` $\ldots$ `,` T_n **where** E", $type(E) = \textbf{bool}$, and for all $i \in 0..n$, $type(V_i) = T_i$.
- A has the form "$G\ O_1\ \ldots\ O_n$" and for each O_i of the form `!`E (respectively of the form `?`P), $type(E)$ (respectively $type(P)$) is defined.
- A has the form "A_1 `;` A_2" and both A_1 and A_2 are well-typed.
- A has the form "**select** A_1 `[]` $\ldots$ `[]` A_n **end**" and each A_i ($i \in 1..n$) is well-typed.
- A has the form "**case** E **is** $P_1 \rightarrow A_1$ `|` $\ldots$ `|` $P_n \rightarrow A_n$ **end**", $type(E)$ is defined, and for all $i \in 1..n$, $type(P_i) = type(E)$ and A_i is well-typed.
- A has the form "**while** E **do** A_0 **end**", $type(E) = \textbf{bool}$, and A_0 is well-typed.

Finally, an NTIF automaton $\langle \mathcal{T}, \mathcal{C}, \mathcal{F}, \mathcal{V}, \mathcal{X}, E_0, \mathcal{G}, \mathcal{S}, s_0, act \rangle$ is well-typed iff $type(E_0) = \textbf{bool}$ and for all $s \in \mathcal{S}$, $act(s)$ is well-typed.

A.3 Variable Initialization

We present an algorithm (see Figure 8) that checks whether variables are systematically defined before used in an NTIF automaton $\langle \mathcal{T}, \mathcal{C}, \mathcal{F}, \mathcal{V}, \mathcal{X}, E_0, \mathcal{G}, \mathcal{S}, s_0, act \rangle$. Its principle is to build two arrays of variable sets named *set_before* and *set_after*, which have the following intuitive meaning:

- *set_before* is a one-dimensional array indexed by $\mathcal{S}$: $set_before[s]$ denotes the set of variables that are necessarily defined before entering state s, given the asumptions that parameters were initially instantiated and that computation started from the initial state.
- *set_after* is a two-dimensional array indexed by $\mathcal{S} \times \mathcal{S}$: $set_after[s][s']$ denotes the set of variables that are necessarily defined after executing an action "**to** s'" in $act(s)$ (if any), given the same asumptions as above.

If *set_after* and *set_before* can be built without raising any error, it follows that all variables of the NTIF process are necessarily defined before used.

To build these arrays, an auxiliary function named *set* is defined (see Figure 7), which takes a set of variables and an action and either returns a one-dimensional array named $after_A$ indexed by $\mathcal{S} \cup \{\textbf{none}\}$, or raises an initialization error. The array $after_A$, returned

by the call $set(before, A)$, contains for each state $s \in \mathcal{S}$ the set of variables necessarily defined when A executes a jump to s (if any), provided the variables in $before_A$ are defined before executing A. Similarly, $after_A[\textbf{none}]$ is the set of variables that are necessarily defined when execution of A terminates normally (without a jump) under the same hypothesis as above.

If no jump to s occurs in A (respectively if A never terminates normally) then $after_A[s]$ (respectively $after_A[\textbf{none}]$) is the set $\mathcal{V}$ of variables of the NTIF automaton[5]. This choice is practical: we assume initially that all variables are initialized by all actions, and then restrict these sets using intersection (this is a greatest fix-point computation). This is theoretically correct since if there is no jump to s or no normal termination then anything can be assumed about the variables that have necessarily been defined in such circumstance.

The *set* and *init* functions use the following auxiliary operators:

- the intersection of variable set arrays is defined by $(a_1 \cap a_2)[s] = a_1[s] \cap a_2[s]$;
- $succ(s)$ is the set of states s' that occur in $act(s)$ (in the form "**to** s'").

The principle of the construction of *set_before* and *set_after* in function *init* (see below) is the following:

- $set_before[s_0]$ is initialized with the set $\mathcal{X}$ of parameters and put in the *visited* set of states.
- For each state s in *visited*, the corresponding $set_after[s]$ array is computed using function *set*, and *set_before* is updated w.r.t. the computed variable sets for the immediately accessible states. *set_before* might be modified for some explored state, in which case the corresponding *set_after* entry must be modified accordingly. This is the reason why some states may be eliminated from *explored* to be put back in *visited*.
- When *visited* is empty, the verification has been done for all the potentially reachable states.

A.4 Unicity of Communication and Next State Reachability

Function *welldef*, defined in Figure 9 (*welldef* stands for *well-defined*), checks whether the two following conditions are satisfied:

- There is at most one communication on each execution path of transitions.
- Every communication is necessarily followed by a jump.

It calls the auxiliary function *reach_to* (defined in Figure 10), which checks that no execution path of the action taken as argument execute a communication and that all its execution paths eventually reach a **to** action.

[5] In practice, *set_after* and $after_A$ should be implemented as sparse matrices of which unspecified elements are interpreted as the set $\mathcal{V}$.

$set(before_A, A) = after_A$ where
for each $s \in \mathcal{S} \cup \{\textbf{none}\}$ do $after_A[s] = \mathcal{V}$ done
case A is
  null $\rightarrow$ $after_A[\textbf{none}] = before_A$
| $V_0, \ldots, V_n$:= $E_0, \ldots, E_n \rightarrow$
    if $\bigcup_{i=0}^{n} use(E_i) \not\subseteq before_A$ then raise error
    else $after_A[\textbf{none}] = before_A \cup \{V_0, \ldots, V_n\}$
| $V_0, \ldots, V_n$:= **any** $T_0, \ldots, T_n$ **where** $E \rightarrow$
    if $use(E) \not\subseteq before_A \cup \{V_0, \ldots, V_n\}$ then raise error
    else $after_A[\textbf{none}] = before_A \cup \{V_0, \ldots, V_n\}$
| **reset** $V_0, \ldots, V_n \rightarrow after_A[\textbf{none}] = before_A \setminus \{V_0, \ldots, V_n\}$
| $G\ O_1\ \ldots\ O_n \rightarrow$
    if $\bigcup_{i=1}^{n} use(O_i) \not\subseteq before_A \cup \bigcup_{i=1}^{n} def(O_i)$ then raise error
    else $after_A[\textbf{none}] = before_A \cup \bigcup_{i=1}^{n} def(O_i)$
| **to** $s \rightarrow after_A[s] = before_A$
| A_1 ; $A_2 \rightarrow$
    $after_{A_1} = set(before_A, A_1)$
    $after_{A_2} = set(after_{A_1}[\textbf{none}], A_2)$
    $after_A = after_{A_1} \cap after_{A_2}$
    $after_A[\textbf{none}] = after_{A_2}[\textbf{none}]$
| **select** A_1 [] $\ldots$ [] A_n **end** $\rightarrow$ $after_A = \bigcap_{i=1}^{n} set(before_A, A_i)$
| **case** E **is** $P_1 \rightarrow A_1$ | $\ldots$ | $P_n \rightarrow A_n$ **end** $\rightarrow$
    if $use(E) \cup \bigcup_{i=1}^{n} use(P_i) \not\subseteq before_A$ then raise error
    else $after_A = \bigcap_{i=1}^{n} set(before_A \cup def(P_i), A_i)$
| **while** E **do** A_0 **end** $\rightarrow$
    if $use(E) \not\subseteq before_A$ then raise error
    else
      $after_{A_0} = set(before_A, A_0)$
      if $before_A \subseteq after_{A_0}[\textbf{none}]$ then
        $after_A = after_{A_0}$
        $after_A[\textbf{none}] = before_A$
      else $after_A = set(before_A \cap after_{A_0}[\textbf{none}], A_0)$

Figure 7: The *set* function.

$init\ \langle \mathcal{T}, \mathcal{C}, \mathcal{F}, \mathcal{V}, \mathcal{X}, E_0, \mathcal{G}, \mathcal{S}, s_0, act \rangle =$
  if $use(E_0) \not\subseteq \mathcal{X}$ then raise error
  foreach $s, s' \in \mathcal{S}$ do
    $set_before[s] = \mathcal{V}$
    $set_after[s][s'] = \mathcal{V}$
  done
  $set_before[s_0] = \mathcal{X}$
  $visited = \{s_0\}$
  $explored = \emptyset$
  while $visited \neq \emptyset$ do
    choose $s \in visited$
    $visited = visited \setminus \{s\}$
    $explored = explored \cup \{s\}$
    $set_after[s] = set(set_before[s], act(s))$
    for each $s' \in succ(s)$ do
      $tmp = set_before[s'] \cap set_after[s][s']$
      if $tmp \neq set_before[s']$ or $s' \notin explored$ then
        $set_before[s'] = tmp$
        $explored = explored \setminus \{s'\}$
        $visited = visited \cup \{s'\}$

Figure 8: The *init* function.

$$
\begin{array}{l}
\mathit{welldef}(A) = \\
\quad \text{case } A \text{ is} \\
\qquad \mathbf{null} \rightarrow \mathbf{true} \\
\qquad \mid \mathbf{null};\ A' \rightarrow \mathit{welldef}(A') \\
\qquad \mid V_0, \ldots, V_n\ \mathtt{:=}\ E_0, \ldots, E_n;\ A' \rightarrow \mathit{welldef}(A') \\
\qquad \mid V_0, \ldots, V_n\ \mathtt{:=}\ \mathbf{any}\ T_0, \ldots, T_n\ \mathbf{where}\ E;\ A' \rightarrow \mathit{welldef}(A') \\
\qquad \mid \mathbf{reset}\ V_0, \ldots, V_n;\ A' \rightarrow \mathit{welldef}(A') \\
\qquad \mid G\ O_1\ \ldots\ O_n;\ A' \rightarrow \mathit{reach_to}(A') \\
\qquad \mid \mathbf{to}\ s;\ A' \rightarrow \mathbf{true}\ (*\ \text{warning might be issued if } A' \neq \mathbf{null}\ *) \\
\qquad \mid (A_1;\ A_2);\ A' \rightarrow \mathit{welldef}(A_1;(A_2;\ A')) \\
\qquad \mid \mathbf{select}\ A_1\ \square\ \ldots\ \square\ A_n\ \mathbf{end};\ A' \rightarrow \bigwedge_{i=1}^{n} \mathit{welldef}(A_i;\ A') \\
\qquad \mid \mathbf{case}\ E\ \mathbf{is}\ P_1 \rightarrow A_1\ \mid\ \ldots\ \mid\ P_n \rightarrow A_n\ \mathbf{end};\ A' \rightarrow \bigwedge_{i=1}^{n} \mathit{welldef}(A_i;\ A') \\
\qquad \mid \mathbf{while}\ E\ \mathbf{do}\ A_0\ \mathbf{end};\ A' \rightarrow \mathit{welldef}(A_0)\ \wedge\ \mathit{welldef}(A') \\
\qquad \mid \text{otherwise} \rightarrow \mathit{welldef}(A;\ \mathbf{null})
\end{array}
$$

Figure 9: Well-definedness of actions.

Function *reach_to* calls the *exh* function (defined in Section A.5) that checks the exhaustivity of a set of patterns.

A.5 Case Exhaustivity

The function checking exhaustivity of a set of patterns uses tuples. A tuple of n elements $Y_1, \ldots, Y_n$ is written $\langle Y_1, \ldots, Y_n \rangle$; in particular the empty tuple is written $\langle\rangle$. The first element of a non-empty tuple is written $\mathit{head}(\vec{Y})$ and the tuple obtained after removing the first element is written $\mathit{tail}(\vec{Y})$. Concatenation of tuples $\vec{Y_1}$ and $\vec{Y_2}$ is written $\vec{Y_1} :: \vec{Y_2}$.

We use tuples of patterns (noted $\vec{P}, \vec{P'}, \ldots$) and tuples of types. The type of a pattern tuple $\vec{P}$, written $\mathit{type}(\vec{P})$, is a type tuple such that $\mathit{type}(\langle\rangle) = \langle\rangle$, $\mathit{head}(\mathit{type}(\vec{P})) = \mathit{type}(\mathit{head}(\vec{P}))$, and $\mathit{tail}(\mathit{type}(\vec{P})) = \mathit{type}(\mathit{tail}(\vec{P}))$.

The case exhaustivity checking function *exh* (defined in Figure 11) takes as input a set $\mathcal{P}$ of pattern tuples, all of the same type, which is written $\mathit{type}(\mathcal{P})$. For convenience, if $\vec{P} = \langle P_1, \ldots, P_n \rangle$ then we write $C(\vec{P})$ instead of $C(P_1, \ldots, P_n)$. For a **case** action the patterns of which are $P_1, \ldots, P_n$, *exh* is initially called with parameter $\{\langle P_1 \rangle, \ldots, \langle P_n \rangle\}$.

The case exhaustivity analysis assumes that execution of guards in patterns can always be false. Consequently, guarded patterns are not taken into account in the analysis since it is not possible to know before the execution whether the guard will be satisfied or not. This algorithm has the same behavior as the one implemented in e.g., CAML or LOTOS NT.

$reach_to(A) =$
  case A is
    null $\rightarrow$ **false**
  | **null**; $A' \rightarrow reach_to(A')$
  | $V_0, \ldots, V_n$:= $E_0, \ldots, E_n$; $A' \rightarrow reach_to(A')$
  | $V_0, \ldots, V_n$:= **any** $T_0, \ldots, T_n$ **where** E; $A' \rightarrow (E =$ **true**$) \ \wedge \ reach_to(A')$
  | **reset** $V_0, \ldots, V_n$; $A' \rightarrow reach_to(A')$
  | $G\ O_1\ \ldots\ O_n$; $A' \rightarrow$ **false**
  | **to** s; $A' \rightarrow$ **true** (* warning might be issued if $A' \neq$ **null** *)
  | $(A_1;\ A_2);\ A' \rightarrow reach_to(A_1;(A_2;\ A'))$
  | **select** A_1 [] $\ldots$ [] A_n **end**; $A' \rightarrow n > 0 \ \wedge \ \bigwedge_{i=1}^{n} reach_to(A_i;\ A')$
  | **case** E **is** $P_1 \rightarrow A_1$ | $\ldots$ | $P_n \rightarrow A_n$ **end**; $A' \rightarrow$
      $exh(\{\langle P_1\rangle, \ldots, \langle P_n\rangle\}) \ \wedge \ \bigwedge_{i=1}^{n} reach_to(A_i;\ A')$
  | **while** E **do** A_0 **end**; $A' \rightarrow$
      if **while** obtained by expansion of **for** then
        $reach_to(A_0;\ A') \ \wedge \ reach_to(A')$
      else **false**
  | otherwise $\rightarrow reach_to(A;$ **null**$)$

Figure 10: Reachability of **to** actions without communications.

$exh(\mathcal{P}) =$
  if $\mathcal{P} = \emptyset$ then **false**
  else if $\mathcal{P} = \{\langle\rangle\}$ then **true**
  else
    for each constructor C of type $head(type(\mathcal{P})) = T_1 \times \cdots \times T_n \rightarrow T$
      let $\mathcal{P}_C = \{\vec{P'} :: tail(\vec{P}) \mid \vec{P} \in \mathcal{P} \ \wedge \ head(\vec{P}) = C(\vec{P'})\} \cup$
        $\{\langle$**any** $T_1, \ldots,$ **any** $T_n\rangle :: tail(\vec{P}) \mid \vec{P} \in \mathcal{P} \ \wedge \ head(\vec{P}) \in \mathcal{V} \cup \{$**any** $T\}\}$
      in $\bigwedge_C \ exh(\mathcal{P}_C)$

Figure 11: Exhaustivity of pattern-matching.

B Dynamic Semantics

B.1 Expression Evaluation

Let the set of values of type T be written $values(T)$ and defined as follows:

$$values(T) = \{v \in Val(\mathcal{T}, \mathcal{C}) \mid type(v) = T\}.$$

For each function symbol F of prototype $T_1 \times \cdots \times T_n \rightarrow T$, we assume a function $[\![F]\!]$ of domain $values(T_1) \times \cdots \times values(T_n) \rightarrow values(T)$.

Expression evaluation is defined as a predicate $eval(E, \rho, v)$, meaning that expression E evaluates to v in store ρ.

$$\rho(V) = v \Rightarrow eval(V, \rho, v)$$

$$eval(E_1, \rho, v_1) \ \wedge \ \ldots \ \wedge \ eval(E_n, \rho, v_n) \Rightarrow eval(C(E_1, \ldots, E_n), \rho, C(v_1, \ldots, v_n))$$

$$eval(E_1, \rho, v_1) \ \wedge \ \ldots \ \wedge \ eval(E_n, \rho, v_n) \Rightarrow eval(F(E_1, \ldots, E_n), \rho, [\![F]\!](v_1, \ldots, v_n))$$

B.2 Pattern-Matching

The function $match(v, \rho, P)$ implements standard pattern matching. If the result is a store ρ', it means that value v matches pattern P in store ρ and that the store ρ is modified into ρ' according to the values assigned to defined variables of P during matching. Otherwise, the result is **fail**, meaning that the value does not match the pattern.

$match(v, \rho, P) =$
  case $\langle v, P \rangle$ is
    $\langle v, V \rangle \rightarrow \rho \oslash [V \mapsto v]$
  | $\langle C(v_1, \ldots, v_n), C(P_1, \ldots, P_n) \rangle \rightarrow$
      let $\rho_1 = \rho$ in
        if $match(v_1, \rho_1, P_1) = \rho_2$ ($\neq$ **fail**) and then
          ...
          and then $match(v_i, \rho_i, P_i) = \rho_{i+1}$ ($\neq$ **fail**) and then
          ...
          and then $match(v_n, \rho_n, P_n) = \rho_{n+1}$ ($\neq$ **fail**) then ρ_{n+1}
        else **fail**
  | $\langle v, P_0$ **where** $E \rangle \rightarrow$
      if $match(v, \rho, P_0) = \rho'$ and then $eval(E, \rho', \mathbf{true})$ then ρ'
      else **fail**
  | $\langle v,$ **any** $T \rangle \rightarrow \rho$
  | otherwise $\rightarrow$ **fail**



Unit´e de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)

Unit´e de recherche INRIA Lorraine : LORIA, Technop^ole de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unit´e de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unit´e de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unit´e de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399