INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

SVL: a Scripting Language for Compositional Verification

Hubert Garavel — Frédéric Lang

N° 4223

Juillet 2001

THÈME 1

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--4223--FR+ENG

SVL: a Scripting Language for Compositional Verification

Hubert Garavel* , Frédéric Lang†

Thème 1 — Réseaux et systèmes
Projet VASY

Rapport de recherche n° 4223 — Juillet 2001 — 36 pages

Abstract: Compositional verification is a way to avoid state explosion for the enumerative verification of complex concurrent systems. Process algebras such as LOTOS are suitable for compositional verification, because of their appropriate parallel composition operators and concurrency semantics. Extending prior work by Krimm and Mounier, this report presents the SVL language, which allows compositional verification of LOTOS descriptions to be performed simply and efficiently. A compiler for SVL has been implemented using an original compiler-generation technique based on the Enhanced LOTOS language. This compiler supports several formats and tools for handling Labeled Transition Systems. It is available as a component of the CADP toolbox and has been applied on various case-studies profitably.

Key-words: Abstraction, Bisimulation, Compositional Verification, Concurrency, Coordination Language, E-LOTOS, Enumerative Verification, Labeled Transition System, LOTOS, Model-Checking, Process Algebra, Reachability Analysis, Specification, Validation.

A short version of this report is also available as "*SVL: a Scripting Language for Compositional Verification*", in Proceedings of the 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems FORTE'2001 (Cheju Island, Korea), August 28-31, 2001.

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

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 Montbonnot-St-Martin (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

SVL : un langage de script pour la vérification compositionnelle

Résumé : La vérification compositionnelle est un moyen d'éviter l'explosion combinatoire d'états lors de la vérification énumérative de systèmes concurrents complexes. Les algèbres de processus telles que LOTOS conviennent bien à la vérification compositionnelle de par leurs opérateurs de composition parallèle et leur sémantique du parallélisme adéquats. Prolongeant les travaux de Krimm et Mounier, ce rapport présente le langage SVL qui permet de vérifier simplement et efficacement des descriptions LOTOS de manière compositionnelle. Un compilateur pour SVL a été réalisé selon une technique originale de construction de compilateurs basée sur le langage *Enhanced* LOTOS. Ce compilateur s'appuie sur divers formats et outils pour représenter et manipuler des systèmes de transitions étiquetées. Il est intégré à la boîte à outils CADP et a été appliqué avec succès à plusieurs études de cas.

Mots-clés : Abstraction, bisimulation, vérification compositionnelle, concurrence, langage de coordination, E-LOTOS, vérification exhaustive, système de transitions étiquetées, LOTOS, vérification basée sur les modèles, algèbre de processus, analyse d'atteignabilité, spécification, validation.

1 Introduction

Enumerative verification (also called *reachability analysis* or *model-checking*) is a popular technique for verifying concurrent systems. Roughly speaking, it is a "brute force" technique, which consists in exploring and checking all states and transitions reachable by a concurrent system. This technique is confronted to the *state explosion* problem, which occurs when the number of states grows exponentially as the number of concurrent processes in the system increases. To avoid or reduce state explosion, various approaches have been proposed, among which: symbolic verification, on-the-fly verification, partial orders, symmetries, data flow analysis, and compositional verification.

This report deals with the latter approach (also known as *compositional reachability analysis* or *compositional minimization*). This approach assumes that the concurrent system under study can be expressed as a collection of communicating sequential processes, the behaviors of which are modeled as finite state machines or labelled transition systems (LTSs, for short). The sequential processes are composed in parallel, either in a flat or hierarchical manner.

In its simplest form [Fer88, MSGS88, SLU89, YY91, TK93a, TK93b, Val93], compositional verification consists in replacing each sequential process by an *abstraction*, simpler than the original process but still preserving the properties to be verified on the whole system. Quite often, abstracting a process is done by minimizing its corresponding LTS modulo an appropriate equivalence or preorder relation (e.g., a bisimulation relation, such as strong, branching or observational equivalence). If the system has a hierarchical structure, minimization can also be applied at every intermediate level in the hierarchy. Clearly, this approach is only possible if the parallel composition is "compatible" with LTS minimization: in particular, this is the case with the parallel composition operators of most process algebras, for which bisimulation is a congruence (see [YY91] for a discussion on this issue).

Although this simple form of compositional verification has been applied successfully to some complex systems (e.g., [FGM+92, CGM+96] in the case of the LOTOS language [ISO88]), it may be counter-productive in some other cases: generating the LTS of each process separately may lead to state explosion, whereas the generation of the whole system of concurrent processes might succeed if processes constrain each other when composed in parallel.

This issue has been addressed by refined compositional verification approaches [GS90, CK93, Yeh93, CK95, CK96, GSL96, KM97, Che98, Gia99], which allow to generate the LTS of each separate process by taking into account *interface constraints* (also known as *environment constraints* or *context constraints*). These constraints express the behavioral restrictions imposed on each process by synchronization with its neighbor processes. Taking into account the environment of each process allows to eliminate states and transitions that are not reachable in the LTS of the whole system. Depending on the approach, interface constraints can be either written by the user or generated automatically.

The refined approach to compositional verification has been implemented in two tools, namely the TRACTA tool [Gia99] and the PROJECTOR/DES2AUT tools [KM97]. The latter tools are part of the CADP protocol engineering toolbox [FGK+96] and have been applied to industrial case-studies [KM97, Pec99, GVZ01]. Although positive, these experiments revealed various issues and shortcomings that prevented compositional verification to be used on a larger scale, especially in industrial projects. To solve these problems, we designed a scripting language named SVL, which can be seen as a process algebra extended with operations on LTSs, e.g., minimization (also called reduction), abstraction, comparison, deadlock/livelock detection, etc. We implemented a compiler for this language, with the goal of making compositional verification easier for non-experts.

This report is organized as follows. Section 2 gives a few preliminary definitions. Section 3 briefly presents the principles and limitations of the DES2AUT tool of [KM97]. Section 4 defines the syntax and semantics of the SVL language. Section 5 introduces high-level features of SVL, which allow sophisticated strategies for compositional verification. Section 6 describes the implementation of the SVL 2.0 compiler. Section 7 gives concluding remarks and lists directions for future work. Finally, Appendices A and B illustrate the benefits of SVL on practical examples.

2 Definitions

Labelled Transition Systems are the natural model for action-based specification languages, especially process algebras such as CCS [Mil89], CSP [Hoa85], ACP [BK84], or LOTOS [ISO88]. Formally, an LTS is a tuple $M = (S, A, T, s_0)$, where S is the set of *states*, A the set of *actions* (or *labels*), $T \subseteq S \times A \times S$ the *transition relation*, and $s_0 \in S$ the *initial state*. A transition $(s, a, s') \in T$ indicates that the system can evolve from state s to state s' by performing action a. In enumerative verification, there are essentially two ways to represent an LTS:

- An *explicit* LTS is defined in extension, by enumerating all its states and transitions. Practically, there exist several formats to store explicit LTSs in computer files. The CADP verification tool set uses three such formats: AUT, a simple textual format, BCG (*Binary Coded Graphs*), a compact binary format based upon dedicated compression algorithms, and SEQ, a human-readable format for displaying sequences of transitions produced by verification tools to explain why a given property is not verified. There exist other LTS formats, for instance the FC2 format used by the FC2TOOLS [BRRd96], with which the CADP tools are interfaced.
- An *implicit* LTS is defined in comprehension by giving its initial state s_0 and its successor function $succ : S \to 2^T$ defined by $succ(s) = \{(s, a, s') \mid (s, a, s') \in T\}$. A generic representation of implicit LTSs is provided by the language-independent environment OPEN/CÆSAR [Gar98] embedded in CADP. OPEN/CÆSAR offers primitives for accessing the initial state of an LTS and for enumerating the successors of a given state, as well as various data structures (state tables, stacks, etc.), allowing on-the-fly

verification algorithms to be implemented concisely. A number of input languages are connected to OPEN/CÆSAR, including LOTOS and the EXP formalism of CADP, which describes implicit LTSs as sets of explicit LTSs combined together using LOTOS parallel composition and hiding operators. Similarly, the FC2 format allows to describe networks of communicating automata, which can be processed by the FC2TOOLS.

The tools available in CADP and FC2TOOLS allow to perform the usual operations on (explicit or implicit) LTSs in several complementary ways, as summarized in the following table:

ALDÉBARAN	Reduction, comparison, deadlock/livelock detection
BCG_IO	Conversion from one explicit LTS format to another
BCG_LABELS	Hiding and renaming of labels
BCG_MIN	Reduction
BCG_OPEN	Implicit LTS view of an explicit LTS
CÆSAR.ADT, CÆSAR	LTS generation from a LOTOS description
CÆSAR.OPEN	Implicit LTS view of a LOTOS description
EVALUATOR	Model-checking, deadlock/livelock detection
EXHIBITOR	Deadlock detection
EXP2FC2	Conversion from EXP format to FC2
EXP.OPEN	Implicit LTS view of an EXP description
FC2EXPLICIT, FC2IMPLICIT	Reduction, comparison, deadlock/livelock detection
GENERATOR	Explicit LTS generation from an implicit LTS
PROJECTOR	Abstraction (see Section 3 below)

3 The Des2Aut tool

DES2AUT is a tool performing compositional generation of an LTS from a composition expression (called *behavior*) written in the DES language [KM97]. Given a behavior, the DES2AUT tool generates and minimizes its LTS compositionally modulo a relation R (e.g., strong or branching bisimulation) specified on the command-line. The DES language is defined by the following grammar, where B, B_0, B_1, B_2 are non-terminal symbols denoting behaviors and F, P, $G_1, \ldots, G_n$ are terminal symbols denoting respectively file prefixes, LOTOS process identifiers, and LOTOS gate identifiers:

$$
\begin{array}{rcll}
B & ::= & F\texttt{.aut} \mid F\texttt{.exp} \mid F\texttt{.lotos} \mid \textbf{Proc}(F\texttt{.lotos}, P\texttt{[}G_1,\ldots,G_n\texttt{]}) & (1)\\
& \mid & \textbf{hide } G_1,\ldots,G_n \textbf{ in } B_0 & (2)\\
& \mid & B_1 \texttt{ |[}G_1,\ldots,G_n\texttt{]| } B_2 & (3)\\
& \mid & B_1 \texttt{ -|[}G_1,\ldots,G_n\texttt{]| } B_2 \mid B_1 \texttt{ -|[}G_1,\ldots,G_n\texttt{]|? } B_2 & (4)
\end{array}
$$

The syntactic constructs above denote:

(1) an explicit LTS, either contained in a file in AUT format, or generated from an implicit LTS given as an EXP description, an entire LOTOS description, or a particular process

P defined in a LOTOS description (this process being instantiated with actual gate parameters $G_1, \ldots, G_n$);

(2) the behavior B_0 in which gates $G_1, \ldots, G_n$ are hidden using the LOTOS hiding operator;

(3) the behaviors B_1 and B_2 executing in parallel with synchronization and communication on gates $G_1, \ldots, G_n$ according to the LOTOS parallel composition semantics;

(4) the behavior B_1 restricted by synchronizing it on the set of gates $G_1, \ldots, G_n$ with B_2 (B_2 is considered as an automaton, the regular language of which expresses interface constraints); the resulting behavior is called the *abstraction* of B_1 w.r.t. B_2. The "`?`" symbol, if present, indicates that the user is unsure that the interface constraints expressed in B_2 are sound w.r.t. the real environment of B_1: as a consequence, validity checks are performed when the resulting LTS is composed in parallel with other LTSs.

Although the usefulness of the DES2AUT tool has been established on significant case-studies [KM97, Pec99, GVZ01], its applicability is limited by several practical shortcomings:

- It only supports a single verification strategy: for each parallel composition operator, the operands are minimized first, then combined in parallel. Therefore, operands can only be combined two by two according to the evaluation order defined by the algebraic parallel composition operators; it is not possible to combine more than two components simultaneously. In some cases, this leads to state explosions that could be avoided otherwise. In the following example,

  `(User1.aut ||| User2.aut) |[`G`]| Medium.aut`

  generating the interleaved combination of "`User1.aut`" and "`User2.aut`" first — independently from "`Medium.aut`" — is likely to produce a larger state space than if "`Medium.aut`" was taken into account.

  So far, this problem has been circumvented by using other CADP tools (namely ALDÉBARAN and EXP.OPEN), which allow several LTSs to be combined together using LOTOS parallel composition operators. Practically, this is tedious because the DES and EXP languages accepted by DES2AUT and ALDÉBARAN–EXP.OPEN, respectively, are incompatible in syntax and semantics, and because many auxiliary DES and EXP files are needed to describe a compositional verification scenario this way. The problem is even worse as there are often many possible scenarios corresponding to various decompositions of the system into sub-systems, which requires a trial-and-error approach to find the most effective decomposition.

- The DES2AUT tool relies on ALDÉBARAN to perform LTS minimization and does not support other bisimulation tools such as BCG_MIN and FC2TOOLS. Moreover, it only accepts the AUT format, which is much less compact than the BCG format, so that compositional verification often aborts due to a lack of disk space.

- The DES2AUT tool has some implementation problems: the DES format parser is overly strict with respect to occurrences of spaces and newlines in certain contexts; the names of intermediate files generated by DES2AUT can exceed the maximal length of filenames allowed by the file system; finally, DES2AUT is based on the UNIX C-shell, which causes portability issues for LINUX and WINDOWS.

- When a compositional verification scenario fails (e.g., if LTS generation, minimization, or parallel composition aborts at some point, because of a lack of memory or disk space, for instance), localizing and understanding the reason of the problem is difficult, as DES2AUT does not provide sufficient debugging information.

4 The SVL language

To address these problems, we propose a scripting language for compositional verification. This language, named SVL, contains all the features of the DES language, as well as new ones that provide increased flexibility in the verification task.

To define the abstract syntax of SVL, we first introduce several terminal symbols: F denotes a file prefix, $G, G_1, \ldots, G_n$ denote LOTOS gate identifiers, $L, L_1, \ldots, L_n$ denote (UNIX-like) regular expressions on labels considered as character strings, and P denotes a LOTOS process identifier. We also introduce the following non-terminal symbols:

$$\begin{array}{rcl}
op & ::= & \texttt{<=} \mid \texttt{==} \mid \texttt{>=} \\
par & ::= & \texttt{|[}G_1, \ldots, G_n\texttt{]|} \mid \texttt{|||} \mid \texttt{||} \\
E & ::= & \texttt{aut} \mid \texttt{bcg} \mid \texttt{exp} \mid \texttt{fc2} \mid \texttt{lotos} \mid \texttt{seq} \\
R & ::= & \textbf{branching} \mid \textbf{strong} \mid \textbf{observational} \mid \textbf{safety} \mid \textbf{tau*.a} \mid \ldots \\
M & ::= & \textbf{std} \mid \textbf{fly} \mid \textbf{bdd} \\
T & ::= & \textbf{aldebaran} \mid \textbf{bcg_min} \mid \textbf{evaluator} \mid \textbf{exhibitor} \mid \textbf{fc2tools} \mid \ldots \\
U & ::= & \textbf{user} \mid <\!\textit{empty}\!>
\end{array}$$

where *op* denotes an equivalence or preorder relation, *par* a LOTOS parallel operator ("$\texttt{|[}G_1, \ldots, G_n\texttt{]|}$" meaning parallel composition with synchronization on gates $G_1, \ldots, G_n$, "`|||`" meaning parallel composition without synchronization, and "`||`" meaning parallel composition with synchronization on all visible gates), E a file extension, R a bisimulation relation, M an algorithmic method to compute bisimulations ("**std**" meaning a standard partition refinement algorithm such as Paige-Tarjan or Kanellakis-Smolka, "**fly**" meaning the Fernandez-Mounier on-the-fly algorithm, and "**bdd**" meaning an algorithm based on Binary Decision Diagrams), and T a tool.

The two main non-terminal symbols are S (*statements*), and B (*behaviors*). They are defined by the following grammar, where '[' and ']' delimit optional clauses in the grammar, and where '`[`' and '`]`' denote the terminal bracket symbols. An SVL program is a list of statements "$S_1; \ldots; S_n$".

$$
\begin{array}{llll}
S & ::= & \texttt{"}F\texttt{.}E\texttt{"} \ \texttt{=} \ B_0 & (S1) \\
 & | & \texttt{"}F\texttt{.}E\texttt{"} \ \texttt{=} \ R \ \textbf{comparison} \ [\textbf{using} \ M] \ [\textbf{with} \ T] \ B_1 \ \mathit{op} \ B_2 & (S2) \\
 & | & \texttt{"}F\texttt{.}E\texttt{"} \ \texttt{=} \ \textbf{deadlock} \ [\textbf{with} \ T] \ \textbf{of} \ B_0 & (S3) \\
 & | & \texttt{"}F\texttt{.}E\texttt{"} \ \texttt{=} \ \textbf{livelock} \ [\textbf{with} \ T] \ \textbf{of} \ B_0 & (S4) \\
 & | & [\texttt{"}F_1\texttt{.}E\texttt{"} \ \texttt{=}] \ \textbf{verify} \ \texttt{"}F_2\texttt{.mcl"} \ \textbf{in} \ B_0 & (S5) \\
B & ::= & \texttt{"}F\texttt{.}E\texttt{"} \ | \ \texttt{"}F\texttt{.lotos"} : P\,\texttt{[}G_1, \ldots, G_n\texttt{]} & (B1) \\
 & | & \textbf{hide} \ [\textbf{all but}] \ L_1, \ldots, L_n \ \textbf{in} \ B_0 & (B2) \\
 & | & B_1 \ \mathit{par} \ B_2 & (B3) \\
 & | & \textbf{rename} \ L_1 \rightarrow L_1', \ldots, L_n \rightarrow L_n' \ \textbf{in} \ B_0 & (B4) \\
 & | & \textbf{generation of} \ B_0 & (B5) \\
 & | & R \ \textbf{reduction} \ [\textbf{using} \ M] \ [\textbf{with} \ T] \ \textbf{of} \ B_0 & (B6) \\
 & | & U \ \textbf{abstraction} \ B_1 \ \textbf{sync} \ G_1, \ldots, G_n \ \textbf{of} \ B_2 & (B7) \\
 & | & B_2 \ \texttt{-|[}G_1, \ldots, G_n\texttt{]|}[\texttt{?}] \ B_1 & (B8)
\end{array}
$$

An Svl behavior denotes either an explicit or an implicit Lts, contrary to the Des language, in which every implicit Lts is immediately converted to an explicit one. Formally, the semantics of a behavior B is given by a denotation function $[\![B]\!]_\sigma$, the result of which is either an (explicit or implicit) Lts file, a Lotos file, a Lotos process instantiation, or an Exp composition expression (i.e., a set of explicit Ltss combined together with hiding and parallel composition operators). The subscript σ denotes a set of file extensions, which denote all acceptable formats in which the result of $[\![B]\!]_\sigma$ should be produced. The value of σ is determined by the context in which B will be used, and namely by the tools that will be applied to B, given that certain tools require certain formats for their inputs and outputs (for instance, the Fc2Tools only handle Ltss in the Fc2 format). Hence, format conversions may be required at some places but, for efficiency reasons, should be avoided as much as possible. σ always contains at least an explicit Lts format; if it contains more than one, a preferred Lts format noted $\mathit{pref}(\sigma)$ is selected in the following preference order: `bcg`, then `aut`, then `fc2`, then `seq`. A dedicated type-checking is done on Svl programs, mainly to distinguish between explicit and implicit Ltss: certain constraints on σ (listed below) must be satisfied, otherwise type-checking errors are reported. The semantics of behaviors is the following:

($B1$) denotes a behavior contained in a file (or a Lotos process instantiation). If it is an implicit Lts file (resp., a Lotos file), then `exp` or `fc2` (resp., `lotos`) must belong to σ. If $E \in \sigma$ then $[\![B]\!]_\sigma$ returns B else it converts the Lts contained in B to $\mathit{pref}(\sigma)$ using Bcg_Io or Exp2Fc2. Lotos process instantiations are handled similarly as Lotos files.

($B2$) denotes the label hiding of an implicit or explicit Lts using the Aldébaran, Bcg_Labels, and Exp.Open tools. This generalizes the Lotos hiding operator by allowing regular expressions on labels and/or by defining the labels not to be hidden (using the "**all but**" keyword). Contrary to the Des language, in which all implicit

LTSs are converted into explicit ones before hiding, SVL semantics preserves implicit LTSs as long as possible (according to the definition of EXP composition expressions). If σ contains `exp` or `fc2` and all labels $L_1, \ldots, L_n$ are LOTOS gates and the "**all but**" keyword is absent and B_0 is a parallel composition of behaviors, then $[\![B]\!]_\sigma$ returns the EXP composition expression **hide** $L_1, \ldots, L_n$ **in** $[\![B_0]\!]_{\{\texttt{exp},\texttt{aut},\texttt{bcg},\texttt{fc2},\texttt{seq}\}}$, else $[\![B]\!]_\sigma$ returns the conversion to *pref*(σ) of $[\![B_0]\!]_{\{\texttt{bcg}\}}$ in which labels matching (respectively not matching if the "**all but**" keyword is present) one of $L_1, \ldots, L_n$ are hidden.

($B3$) denotes the parallel composition of B_1 and B_2. σ must contain `exp` or `fc2`. $[\![B]\!]_\sigma$ returns the EXP composition expression $[\![B_1]\!]_{\sigma'}$ *par* $[\![B_2]\!]_{\sigma'}$ where $\sigma' = \{\texttt{exp}, \texttt{aut}, \texttt{bcg}, \texttt{fc2}, \texttt{seq}\}$.

($B4$) denotes the label renaming of an explicit LTS using the BCG_LABELS tool, which supports UNIX-like regular expression matching and substring replacement. $[\![B]\!]_\sigma$ returns the conversion to *pref*(σ) of $[\![B_0]\!]_{\{\texttt{bcg}\}}$ in which labels are renamed as specified by the rules $L_1 \rightarrow L'_1, \ldots, L_n \rightarrow L'_n$.

($B5$) denotes the conversion from an implicit LTS to an explicit one, which is computed using OPEN/CÆSAR compilers and GENERATOR. In SVL, such conversions must be requested explicitly using the "**generation**" operator, unlike the DES language in which such conversions are automatic and cannot be avoided. Let α be $[\![B_0]\!]_{\sigma \cup \{\texttt{exp},\texttt{fc2},\texttt{lotos}\}}$. If α is already an explicit LTS, then $[\![B]\!]_\sigma$ returns α else it returns the conversion of α to an explicit LTS of format *pref*(σ).

($B6$) denotes the reduction of an LTS modulo an equivalence relation R, using an algorithmic method M and a tool T (ALDÉBARAN, BCG_MIN, or FC2TOOLS). For short, we abbreviate this operator to *RMT-reduction*. $[\![B]\!]_\sigma$ returns the conversion to format *pref*(σ) of the RMT-reduction of $[\![B_0]\!]_{\sigma^T}$, where σ^T is the set of input formats accepted by tool T (see Figure 1 for the definition of σ^T). The parameters M and T are optional; by default, the BCG_MIN tool and the "**std**" method are selected.

($B7$) denotes the abstraction w.r.t. interface constraints, which is computed using OPEN/CÆSAR compilers and PROJECTOR. $[\![B]\!]_\sigma$ returns the conversion to format *pref*(σ) of the abstraction of $[\![B_2]\!]_{\{\texttt{bcg},\texttt{exp},\texttt{fc2},\texttt{lotos}\}}$ w.r.t. the interface $[\![B_1]\!]_{\{\texttt{aut}\}}$. The "**user**" keyword indicates that the interface is provided by the user and that the correctness of this interface must be checked as explained in [KM97].

($B8$) is an alternative notation for abstraction, reminiscent of the DES language and kept for compatibility. The optional "**?**" symbol has the same meaning as the "**user**" keyword in ($B7$). Compared to ($B7$), operands B_1 and B_2 appear in reverse order, B_1 being the interface.

The statements have the following effects:

T	σ^T
ALDÉBARAN	{aut, bcg, exp, fc2, seq}
BCG_MIN	{bcg}
EVALUATOR	{bcg, exp, fc2, lotos}
EXHIBITOR	{bcg, exp, fc2, lotos}
FC2TOOLS	{fc2}

Figure 1: Input formats accepted by the different tools.

($S1$) stores in file F.E (where $E \neq$ `lotos`) either $[\![B_0]\!]_{\{E\}}$ if E denotes an explicit LTS format or $[\![B_0]\!]_{\{\mathtt{aut},\mathtt{bcg},\mathtt{exp},\mathtt{fc2},\mathtt{seq}\}}$ if E denotes an implicit LTS format.

($S2$) compares two LTSs modulo an equivalence or preorder relation R, using an algorithmic method M and a tool T (ALDÉBARAN or FC2TOOLS). Formally, it compares $[\![B_1]\!]_{\sigma^T}$ and $[\![B_2]\!]_{\sigma^T}$, where σ^T is defined on Figure 1. The result stored in file F.E (where $E \notin \{\mathtt{exp}, \mathtt{lotos}\}$) is a (set of) distinguishing path(s) if the comparison returns false or an empty path otherwise. The parameters M and T are optional; by default, ALDÉBARAN and the "**std**" method are selected.

($S3$) detects deadlocks using a tool T (ALDÉBARAN, EVALUATOR, EXHIBITOR, or FC2TOOLS) in $[\![B_0]\!]_{\sigma^T}$. The result stored in file F.E (where $E \notin \{\mathtt{exp}, \mathtt{lotos}\}$) is a (set of) path(s) leading to (one or all) deadlock state(s), if any, or an empty path otherwise. If no tool T is specified, then ALDÉBARAN is chosen by default.

($S4$) checks for livelocks in a way similar to statement ($S3$). If no tool T is specified, then ALDÉBARAN is chosen by default.

($S5$) evaluates using the EVALUATOR tool [MS00] a temporal logic formula on $[\![B_0]\!]_{\sigma^{\mathrm{EVALUATOR}}}$. The temporal logic formula is written in regular alternation-free mu-calculus and is contained in file F_2.`mcl`. The EVALUATOR tool returns a truth value (*true* or *false*) and possibly a diagnostic (example or counter-example) explaining this truth value. This diagnostic is a portion of the whole LTS of B_0 and can be stored in file F_1.E (where $E \notin \{\mathtt{exp}, \mathtt{lotos}\}$).

5 Meta-operations

SVL has so-called *meta-operations*, which allow various compositional reduction strategies to be written concisely; this has proven to be useful in many case-studies. Meta-operations extend the syntax of SVL behaviors as follows:

$$B \quad ::= \quad \ldots$$
$$\quad | \quad A\ R\ \textbf{reduction using}\ M\ \textbf{with}\ T\ \textbf{of}\ B_0$$

where the attribute A is defined by:

$$A \quad ::= \quad \textbf{leaf} \mid \textbf{root leaf} \mid \textbf{node}$$

Informally, meta-operations have the effect of propagating RMT-reductions automatically at various places in the algebraic term B_0. Depending on the value of A, they have different semantics:

- "**leaf**" means that an RMT-reduction must be applied to all leaves "F.E" but also to all subterms of B_0 that generate an explicit LTS (i.e., to all abstraction, renaming, hiding, generation, and reduction operators). Leaf reduction is not propagated through generation and reduction operators. Moreover, to maximize the use of the EXP format of CADP and to avoid converting EXP descriptions into explicit LTSs (unless requested by the user), RMT-reduction is not applied to hiding operators whose operands are parallel compositions of LTSs.
- "**root leaf**" is similar to "**leaf**" except that B_0 itself is also RMT-reduced at the top-level.
- "**node**" is similar to "**leaf**" except that RMT-reductions are also applied to all parallel composition operators in the abstract tree of B_0. This emulates the reduction strategy of the DES2AUT tool and is mainly implemented for backward compatibility purpose.

Formally, "A R **reduction using** M **with** T **of** B" is expanded into $\mathcal{E}_{RMT}(B, A)$ where $\mathcal{E}_{RMT}$ is defined as follows and where the shorthand notation $\mathcal{R}_{RMT}(B)$ stands for an RMT-reduction of B:

$\mathcal{E}_{RMT}(B, A) =$ **if** $A =$ "**root leaf**" **then** $\mathcal{R}_{RMT}(\mathcal{E}_{RMT}(B, \textbf{leaf}))$
  else case B **in**
    rename ... **in** $B_0 \rightarrow \mathcal{R}_{RMT}($**rename**... **in** $\mathcal{E}_{RMT}(B_0, A))$
    | U **abstraction** B_1 ... **of** $B_2 \rightarrow$
      $\mathcal{R}_{RMT}(U$ **abstraction** B_1 ... **of** $\mathcal{E}_{RMT}(B_2, A))$
    | B_1 *par* $B_2 \rightarrow$
      if $A =$ "**node**" **then** $\mathcal{R}_{RMT}(\mathcal{E}_{RMT}(B_1, A)$ *par* $\mathcal{E}_{RMT}(B_2, A))$
      else $\mathcal{E}_{RMT}(B_1, A)$ *par* $\mathcal{E}_{RMT}(B_2, A)$
    | **hide** $L_1, \ldots, L_n$ **in** $B_0 \rightarrow$ **if** $\mathcal{E}_{RMT}(B_0, A)$ *has the form* "B_1 *par* B_2" *and*
      all L_i *are* LOTOS *gates* **then hide** $L_1, \ldots, L_n$ **in** $\mathcal{E}_{RMT}(B_0, A)$
      else $\mathcal{R}_{RMT}($**hide** $L_1, \ldots, L_n$ **in** $\mathcal{E}_{RMT}(B_0, A))$
    | **otherwise** $\rightarrow \mathcal{R}_{RMT}(B)$

After meta-operation expansion, all behaviors containing meta-operations are replaced by simple behaviors (as defined in Section 4). Since the application of $\mathcal{E}_{RMT}$ may create superfluous operations, an optimization function called $\mathcal{O}$ is applied to the resulting behaviors in order to increase efficiency. For instance, $\mathcal{O}$ merges nested hiding operators into a single hiding operator. $\mathcal{O}$ also replaces $\mathcal{R}_{RMT}(\mathcal{R}_{R'M'T'}(B))$ by $\mathcal{R}_{R'M'T'}(B)$ if the relation R' is weaker or equal to the relation R (noted $R' \sqsubseteq R$), since an RMT-reduction is useless after an $R'M'T'$-reduction. At last, $\mathcal{O}$ suppresses an RMT-reduction applied to the operand of a hiding (resp. abstraction) operator if the result of this hiding (resp. abstraction) itself is to be reduced modulo the same relation R.

$$
\begin{array}{l}
\mathcal{O}(B) = \textbf{case } B \textbf{ in} \\
\quad \texttt{"}F\texttt{.}E\texttt{"}[\texttt{:}P\texttt{[...]}] \rightarrow B \\
\quad \mid B_1 \ \textit{par} \ B_2 \rightarrow \mathcal{O}(B_1) \ \textit{par} \ \mathcal{O}(B_2) \\
\quad \mid \textbf{generation of } B_0 \rightarrow \textbf{generation of } \mathcal{O}(B_0) \\
\quad \mid \textbf{rename } \ldots \textbf{ in } B_0 \rightarrow \textbf{ rename } \ldots \textbf{ in } \mathcal{O}(B_0) \\
\quad \mid U \textbf{ abstraction } B_1 \ldots \textbf{ of } B_2 \rightarrow U \textbf{ abstraction } \mathcal{O}(B_1) \ldots \textbf{ of } \mathcal{O}(B_2) \\
\quad \mid \textbf{hide } L_1, \ldots, L_n \textbf{ in } B_0 \rightarrow \\
\qquad \textbf{if } B_0 \ \textit{has the form} \text{ ``}\textbf{hide } L_{n+1}, \ldots, L_{n+m} \textbf{ in } B_0'\text{''} \textbf{ then} \\
\qquad\quad \mathcal{O}(\textbf{hide } L_1, \ldots, L_{n+m} \textbf{ in } B_0') \\
\qquad \textbf{else hide } L_1, \ldots, L_n \textbf{ in } \mathcal{O}(B_0) \\
\quad \mid \mathcal{R}_{RMT}(B_0) \rightarrow \textbf{case } B_0 \textbf{ in} \\
\qquad \mathcal{R}_{R'M'T'}(B_0') \ \textit{where} \ R' \sqsubseteq R \rightarrow \mathcal{O}(\mathcal{R}_{R'M'T'}(B_0')) \\
\qquad \mid \textbf{hide } L_1, \ldots, L_n \textbf{ in } \mathcal{R}_{R'M'T'}(B_0') \ \textit{where} \ R = R' \rightarrow \\
\qquad\quad \mathcal{O}(\mathcal{R}_{RMT}(\textbf{hide } L_1, \ldots, L_n \textbf{ in } B_0')) \\
\qquad \mid U \textbf{ abstraction } B_1 \ldots \textbf{ of } \mathcal{R}_{R'M'T'}(B_2) \ \textit{where} \ R = R' \rightarrow \\
\qquad\quad \mathcal{O}(\mathcal{R}_{RMT}(U \textbf{ abstraction } B_1 \ldots \textbf{ of } B_2)) \\
\qquad \mid \textbf{otherwise} \rightarrow \mathcal{R}_{RMT}(\mathcal{O}(B_0))
\end{array}
$$

6 The SVL 2.0 compiler

A compiler for the SVL language has been implemented within the CADP toolbox. This compiler, named SVL 2.0, includes five phases, as shown in Figure 2. It was developed using an original compiler construction technology also used for two other compilers developed by the VASY team of INRIA:

- The lexical analysis, syntax analysis, and abstract tree construction phases (1,100 lines of code) are implemented using the SYNTAX compiler generator [BD97]. SYNTAX has similar functionalities as LEX and YACC, enhanced with a well-designed automatic error recovery mechanism.
- Type checking, expansion of meta-operations, and code generation (2,900 lines of code) are implemented as a set of data types and functions written in the

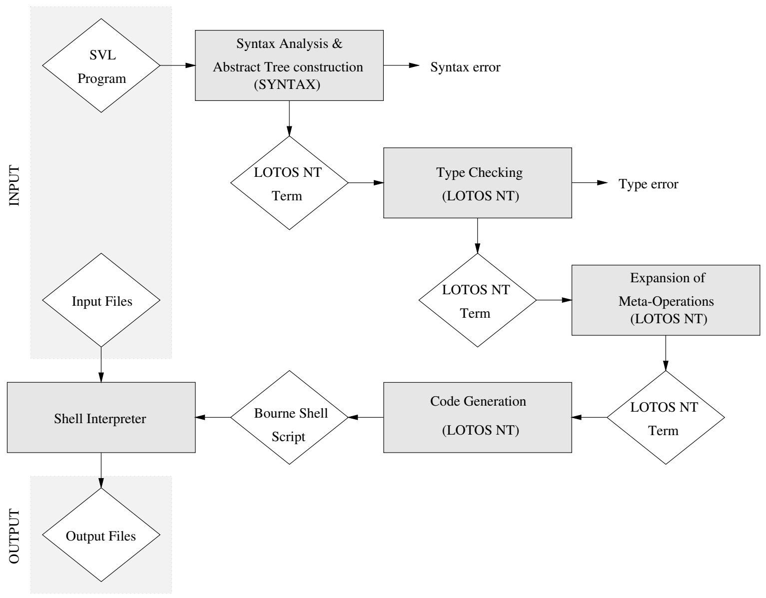

Figure 2: Architecture of SVL 2.0

LOTOS NT language [GS98, Sig00]. Inspired by the standardization work on Enhanced LOTOS [Que01], LOTOS NT combines the theoretical foundations of process algebras with features borrowed from both functional and imperative languages (such as abstract data types, patterns, assignments, loops, ...) suitable for a wider industrial acceptance of formal methods. The TRAIAN compiler [SBC+00] translates the LOTOS NT code into a set of C types and functions. The generated C code is augmented with about 200 lines of hand-written C code.

- SVL generates a script written in Bourne shell. This script starts by including a set of predefined functions (1,750 lines of Bourne shell), then the SVL statements and expressions are translated into sequences of calls to these functions with suitable parameters.

In total, the development of SVL 2.0 took about 5 person · month, and totalizes more than 11,000 lines of generated C code. The SVL compiler is designed carefully to ease the verification task. We give five examples of such a careful design:

- When executing a verification scenario, SVL 2.0 produces simultaneously two kinds of output: a high-level, concise execution trace expressed at the abstraction level of the SVL source program, and a low-level log file which lists all the executed shell commands and the output of their execution. This file is very convenient to locate and understand run-time errors.

- During its execution, the generated script produces several temporary files (explicit LTSs, EXP files, hiding and renaming files, etc.) needed by the various tools of CADP. To minimize disk space consumption, SVL removes temporary files as soon as possible and uses as much as possible the BCG format for representing explicit LTSs, because the BCG format is more compact than other LTS formats.
- Several convenient compiler options are implemented, e.g., "`-debug`", which prevents intermediate files from being erased, "`-expand`", which produces the SVL program obtained by expanding all meta-operations, and "`-script`", which generates the Bourne shell script without executing it.
- As much as possible, the generated script tries to recover from errors automatically by using "expert" knowledge about the verification process. For instance:
  - When a reduction fails because the combination of relation, tool, and method specified in the source SVL program is not available, SVL 2.0 attempts to change some of the parameters (tool and/or method). For instance, it can replace "**std**" with "**bdd**" when using ALDÉBARAN to perform strong reduction of an LTS in the EXP format since standard reduction is not available in this case.
  - If a weak reduction fails, e.g., because of memory exhaustion, SVL 2.0 tries to perform first stronger reduction(s) before trying again the weak reduction. For instance, observational reduction may be preceded by branching reduction, and branching reduction may be preceded by strong reduction. If the reduction still fails, SVL 2.0 will leave the LTS un-reduced and continue execution.
  - The SVL semantics states that a "**generation**" operator is mandatory whenever an implicit LTS should be converted to an explicit LTS because such conversion is costly and can lead to state explosion. Practically, this constraint is slightly relaxed: if a "**generation**" operator is omitted, it will be inserted automatically and a warning message will be emitted.
- SVL 2.0 permits to invoke shell commands from the SVL description. This can be used for calling tools with features not implemented in SVL, for using the shell control structures to perform conditionals and loops, and for modifying the values of environment variables specifying the default tools, methods, and relations as well as options to be passed to the tools.

7 Conclusion

Although compositional verification has a strong potential in attacking the state explosion problem, only a few implementations have been carried out. In this respect, the compositional verification tools of CADP are of particular interest because they are widely distributed and have been applied to several industrial case-studies [CGM+96, KM97, Pec99, GVZ01]. However, these tools require a good level of expertise to be effective. To address this problem, we designed the scripting language SVL, which is well-adapted to compositional verification:

- SVL combines process algebra operators (parallel composition, label hiding) with operations on LTSs (e.g., minimization, abstraction, comparison, livelock/deadlock detection, label hiding and label renaming using regular expressions, etc.). It also provides high-level meta-operators, which allow sophisticated compositional verification strategies to be expressed concisely. Practically, SVL is expressive enough to supersede the two formats EXP and DES previously used in the CADP toolbox and to suppress the need for hand-written MAKEFILES.

- SVL behaves as a tool-independent coordination language (in the same way as EUCALYPTUS [Gar96] is a tool-independent graphical user interface). Due to its concise syntax and well-chosen default options, SVL relieves the user from the burden of launching verification tools manually: it invokes the relevant tools with appropriate options and allocates/deletes temporary files automatically. SVL supports several verification tools (ALDÉBARAN, BCG_MIN, FC2TOOLS) both for *explicit* (enumerative) and *implicit* (on-the-fly) verification. It supports several LTS formats (AUT, BCG, FC2, SEQ) though using as much as possible the BCG format, which allows significant savings in both disk space and access time. Switching from one tool or one format to another can be done simply by changing a few words in the SVL description.

- SVL is extensible in two ways. Its modular design will allow new tools and new formats to be integrated easily. Moreover, as Bourne shell commands can be directly invoked from an SVL description, the user can easily describe specific processings and benefit from the high-level constructs of the Bourne shell (**if** and **case** conditions, **while** and **for** loops, etc.).

For this language, we have fully implemented the SVL 2.0 compiler which is developed using an original approach to compiler construction, combining the SYNTAX compiler generator and LOTOS NT, a variant of the forthcoming E-LOTOS standard.

SVL 2.0 has reached a good level of stability and maturity. It is distributed as a component of CADP and available on four different UNIX and WINDOWS platforms. It has been used in ten case-studies profitably, in particular for the compositional verification of a dynamic reconfiguration protocol for agent-based applications [CGMdP01]. As regards future work, three directions can be identified:

- SVL could be enhanced with common sub-expressions detection. At present, the user can always avoid redundant computations by storing their results in intermediate, named files, which can be reused later. For instance, the statement:

$$
\begin{array}{lcl}
\texttt{"}a\texttt{.bcg"} & \texttt{=} & \textbf{leaf strong reduction of} \\
 & & ((B_1 \;\texttt{|[}G\texttt{]|}\; B_0) \;\texttt{||}\; (B_2 \;\texttt{|[}G\texttt{]|}\; B_0))
\end{array}
$$

can be evaluated more efficiently as:

$$
\begin{array}{lcl}
\texttt{"}b\texttt{.bcg"} & \texttt{=} & \textbf{strong reduction of } B_0 \\
\texttt{"}a\texttt{.bcg"} & \texttt{=} & ((\textbf{strong reduction of } B_1) \;\texttt{|[}G\texttt{]|}\; \texttt{"}b\texttt{.bcg"}) \;\texttt{||} \\
 & & ((\textbf{strong reduction of } B_2) \;\texttt{|[}G\texttt{]|}\; \texttt{"}b\texttt{.bcg"})
\end{array}
$$

Ideally, this optimization could also be performed automatically.

- The SVL language could be enriched with additional operators, e.g., to model-check temporal logic formulas, and additional meta-operators, such as recursive propagation of hiding (so as to hide labels as soon as possible) or recursive abstractions. More applications are needed to determine which extensions are practically useful.
- The SVL language and related tools should be extended to support the new parallel composition operators [GS99] introduced in E-LOTOS and LOTOS NT; these operators are more expressive and user-friendly than LOTOS ones and would thus contribute to make compositional verification easier.

Acknowledgements

We would like to thank the former and current members of the INRIA/VASY team involved in the SVL project, namely: Mark Jorgensen and Christophe Discours, who implemented the early versions (1.0–1.6) of the SVL compiler, Ghassan Chehaibar and Charles Pecheur, who used and gave valuable feedback about the DES2AUT and SVL 1.* tools, Marc Herbert and Stéphane Martin, who tested the improved CADP tools on several architectures carefully, and Radu Mateescu, who experimented the SVL 2.0 compiler on several case-studies.

We are also grateful to Ji He (University of Stirling) for her reports about the DES2AUT tool, to Laurent Mounier (Univ. Joseph Fourier, Grenoble) for his pertinent explanations regarding DES2AUT in particular and compositional verification in general, and to Solofo Ramangalahy, and to the anonymous referees for their useful comments about this report.

A SVL examples

The SVL language and compiler have been used in 19 of the 29 demo examples packaged in the CADP tool kit distribution. This appendix attempts at convincing the reader that SVL allows to write non trivial verification scenarios simply. To this aim, we present excerpts that illustrate some significant features of SVL.

A.1 Compositional verification

This example deals with the compositional verification of a LOTOS specification of the Alternating Bit Protocol [Gar89]. This protocol is divided into four processes. For each process, the corresponding LTS is generated and reduced modulo strong bisimulation. The four LTSs are then composed together (using LOTOS parallel composition and hiding operators) and, finally, the resulting LTS is reduced modulo strong bisimulation.

In SVL, this scenario is written as follows (the "%" symbol is used to mark the beginning of a line containing a Bourne shell command):

```
% DEFAULT_LOTOS_FILE="bitalt_protocol.lotos"

"bitalt_protocol.bcg" = root leaf strong reduction of
    hide SDT, RDT, RDTe, RACK, SACK, SACKe in
        (
            (
                BODY_TRANSMITTER
                |||
                BODY_RECEIVER
            )
            |[SDT, RDT, RDTe, RACK, SACK, SACKe]|
            (
                MEDIUM1
                |||
                MEDIUM2
            )
        );
```

In this example, one can see two reasons for the conciseness of SVL:

- by setting the shell variable "`DEFAULT_LOTOS_FILE`" to the name of the LOTOS file in which the four processes are defined, one avoids to specify this file name before each process name;
- by using the "`root leaf strong reduction`" meta-operation, one avoids to write the following equivalent, but longer, expanded description:

```
"bitalt_protocol.bcg" = strong reduction of
    hide SDT, RDT, RDTe, RACK, SACK, SACKe in
        (
            (
                (strong reduction of BODY_TRANSMITTER)
                |||
                (strong reduction of BODY_RECEIVER)
            )
            |[ SDT, RDT, RDTe, RACK, SACK, SACKe ]|
            (
                (strong reduction of MEDIUM1)
                |||
                (strong reduction of MEDIUM2)
            )
        );
```

When a network of communicating LTSs must be used several times, it is possible to store it in an EXP file:

```
% DEFAULT_LOTOS_FILE="bitalt_protocol.lotos"

"bitalt_protocol.exp" = leaf strong reduction of
    hide SDT, RDT, RDTe, RACK, SACK, SACKe in
        (
            (
                BODY_TRANSMITTER
                |||
                BODY_RECEIVER
            )
            |[SDT, RDT, RDTe, RACK, SACK, SACKe]|
            (
                MEDIUM1
                |||
                MEDIUM2
            )
        );
```

This EXP file can be reused later in various verification scenarios, for instance to check deadlock freeness:

```
"bitalt_dead.seq" = deadlock of "bitalt_protocol.exp";
```

or to perform an on-the-fly comparison with an explicit LTS:

```
"bitalt_oequ.seq" = observational comparison using fly
                    "bitalt_protocol.exp" == "bitalt_service.bcg";
```

Of course, the EXP description can still be used for explicit LTS generation:

```
"bitalt_protocol.bcg" = generation of "bitalt_protocol.exp";
```

A.2 Compositional verification with abstractions

The verification of the rel/REL protocol [BM91, FGM+92] illustrates how interfaces can be used to avoid state explosion. The verification of this protocol using the DES2AUT tool was presented in [KM97]. We present here the same verification using SVL.

The rel/REL example is composed of four processes: one crash transmitter and three receivers. Since the LTSs corresponding to the receivers are too large to be generated directly, the following strategy is adopted:

- The crash transmitter is generated, then reduced modulo strong bisimulation.
- For each receiver, a restricted LTS is generated by taking into account interface constraints provided by the three other processes. These interface constraints are specified in LOTOS by the user; the constraints for receiver i are contained in a file named "r*i*_interface.lotos".
- Restricted LTSs associated to receivers are then reduced modulo strong bisimulation and composed together, each parallel composition being further restricted with respect to the interface constraints of the crash transmitter. The resulting LTS is then reduced modulo strong bisimulation.
- Finally, the LTS associated to the crash transmitter is generated, reduced modulo strong bisimulation, and composed with the LTS resulting from the composition of the receivers. The correcteness of user-given interfaces is checked during this last composition step.

This generation scenario is written in SVL as follows:

```
% DEFAULT_LOTOS_FILE="rel_rel.lotos"

"crash_trans.bcg" = strong reduction of CRASH_TRANSMITTER ;

"rel_rel.bcg" = strong reduction of generation of leaf strong reduction of
    hide R_T1, R_T2, R_T3, R12, R13, R21, R23, R31, R32  in
        (
            (
                (
                    (RECEIVER_NODE_1 -||? "r1_interface.lotos")
                    |[R12, R21, R13, R31]|
                    (
```

```
                    (RECEIVER_NODE_2 -||? "r2_interface.lotos")
                    |[R23, R32]|
                    (RECEIVER_NODE_3 -||? "r3_interface.lotos")
                ) -|[R_T2, R_T3]| "crash_trans.bcg"
            ) -|[R_T1, R_T2, R_T3]| "crash_trans.bcg"
        )
        |[R_T1, R_T2, R_T3]|
        "crash_trans.bcg"
    );
```

Again, the conciseness provided by meta-operators is noticeable. Indeed, expanding the above SVL description would result in a much longer SVL description:

```
% DEFAULT_LOTOS_FILE="rel_rel.lotos"

"crash_trans.bcg" = strong reduction of CRASH_TRANSMITTER ;

"rel_rel.bcg" = strong reduction of generation of
  hide R_T1, R_T2, R_T3, R12, R13, R21, R23, R31, R32  in
    (
      (
        strong reduction of
          (
            (
              (
                strong reduction of
                  (RECEIVER_NODE_1 -||? "r1_interface.lotos")
              )
              |[R12, R21, R13, R31]|
              (
                strong reduction of
                  (
                    (
                      (
                        strong reduction of
                          (RECEIVER_NODE_2 -||? "r2_interface.lotos")
                      )
                      |[R23, R32]|
                      (
                        strong reduction of
                          (RECEIVER_NODE_3 -||? "r3_interface.lotos")
                      )
                    ) -|[R_T2, R_T3]| "crash_trans.bcg"
                  )
              )
            ) -|[R_T1, R_T2, R_T3]| "crash_trans.bcg"
          )
```

```
    )
    |[R_T1, R_T2, R_T3]|
    (strong reduction of "crash_trans.bcg")
  );
```

A.3 Verification using bisimulations

SVL is also convenient to perform verification using bisimulations, as illustrated by the verification of a Plain Ordinary Telephone Service (POTS) tackled by Patrick Ernberg and Laurent Mounier. The LTS corresponding to the formal description in LOTOS of the POTS is first generated non-compositionally, then reduced modulo strong bisimulation:

```
"pots.bcg" = strong reduction of generation of "pots.lotos";
```

Then, several requirements can be checked using bisimulations. We give two examples:

- *It should always be possible to perform the "`S !1 !DIALT !ON`" action from any state.* The verification of this requirement is done by hiding all labels but "`S !1 !DIALT !ON`", and comparing modulo branching equivalence the resulting LTS to an LTS (contained in file "`r2_1.aut`") having a single state and a single cyclic transition labelled with "`S !1 !DIALT !ON`":

```
"diag_2_1_1.seq" = branching comparison using fly
    (total hide all but "S !1 !DIALT !ON" in "pots.bcg") == "r2_1.aut";
```

- *It should always be possible for subscriber 1 to perform an "`ONH`" (onhook) or an "`OFFH`" (offhook) action.* This requirement is verified by renaming all onhook and offhook actions to a unique "`ONHOOK_OFFHOOK`" action and all the other actions to an "`OTHERS`" action, then by checking that the resulting graph is strongly equivalent to the LTS (contained in file "`r4.aut`") having one state and two cyclic transitions respectively labelled with "`ONHOOK_OFFHOOK`" and "`OTHERS`":

```
"diag_4_1.seq" = strong comparison
    (
        total rename
            "S !1 !OFFH !ASUBSC" -> "ONHOOK_OFFHOOK",
            "S !1 !OFFH !BSUBSC" -> "ONHOOK_OFFHOOK",
            "S !1 !ONH" -> "ONHOOK_OFFHOOK",
            ".*" -> "OTHERS"
        in "pots.bcg"
    )
    == "r4.aut";
```

A.4 Parameterized verification scripts

By combining SVL with Bourne shell features (essentially variables and control structures), it is possible to introduce parameterization in verification scenarios.

The first example illustrates the use of a "`for`" loop to verify eight temporal logic properties (contained in files "`prop1.mcl`", "`prop2.mcl`", ..., "`prop8.mcl`") on an LTS generated from a LOTOS description:

```
"bitalt.bcg" = generation of "bitalt.lotos";

% for N in 1 2 3 4 5 6 7 8
% do
   verify "prop$N.mcl" in "bitalt.bcg";
% done
```

Note that this verification could also be done on-the-fly on the LOTOS description itself:

```
% for N in 1 2 3 4 5 6 7 8
% do
   verify "prop$N.mcl" in "bitalt.lotos";
% done
```

A "`for`" loop can also be used to repeat more complex SVL statements. In the following example, each LOTOS file is translated to an LTS, which is then minimized modulo observational equivalence:

```
% for PARAMS in 1-1 1-2 1a-1 1a-2 1a-3 2-1 3-1 5-1 6-1
% do
    "co4-${PARAMS}_omin.bcg" = observational reduction of
                               generation of "co4-${PARAMS}.lotos";
% done
```

Other Bourne shell control structures than "`for`" loops can be used. The following example illustrates a combination of "`for`" and "`if`": files "`a0.bcg`", "`a1.bcg`", "`a2.bcg`", "`b0.bcg`", "`b1.bcg`, "`b2.bcg`" are generated and reduced in turn; every time a "b*i*`.bcg`" file is generated, it is compared to "a*i*`.bcg`":

```
% for P in a b
% do
   % for N in 0 1 2
   % do
      "$P$N.bcg" = strong reduction of generation of "$P$N.lotos";
      % if test $P = b
      % then
         "a$N-b$N_sequ.seq" = safety comparison "a$N.bcg" == "b$N.bcg";
      % fi
```

```
    % done
% done
```

Bourne shell functions can also be used in place of "`for`" loops to provide another form of parameterization. In the following example taken from the verification of various leader election protocols [GM97], there are several files named "`EXPERIMENT_`*i*`.lotos`". Each of them contains six processes named "`STATION_1`", `STATION_2`", "`STATION_3`", "`LINK_1`", "`LINK_2`", and "`LINK_3`", which are composed together (using parallel composition and hiding) and compared on-the-fly modulo branching equivalence to an LTS named "`SERVICE_WITHOUT_CRASHES.bcg`":

```
% for N in 01 02 03 04
% do
    % DEFAULT_COMPARISON_METHOD="fly"
    % DEFAULT_LOTOS_FILE="EXPERIMENT_${N}.lotos"
    "EXPERIMENT_$N.seq" = branching comparison
        leaf strong reduction of
            hide PRED1, SUCC1, PRED2, SUCC2, PRED3, SUCC3 in
                (
                    (
                        STATION_1
                        |||
                        STATION_2
                        |||
                        STATION_3
                    )
                    |[PRED1, SUCC1, PRED2, SUCC2, PRED3, SUCC3]|
                    (
                        LINK_1
                        |||
                        LINK_2
                        |||
                        LINK_3
                    )
                )
        ==
            "SERVICE_WITHOUT_CRASHES.bcg" ;
% done
```

Alternatively, in the same example, a Bourne shell function "`F`" could have been defined to enclose the SVL statements contained in the body of the "`for`" loop. Then, function "`F`" can be invoked several times with different arguments:

```
% F () {
    % DEFAULT_COMPARISON_METHOD="fly"
    % DEFAULT_LOTOS_FILE="EXPERIMENT_$1.lotos"
```

```
    "EXPERIMENT_$1.seq" = branching comparison
        leaf strong reduction of
            hide PRED1, SUCC1, PRED2, SUCC2, PRED3, SUCC3 in
                (
                    (
                        STATION_1
                        |||
                        STATION_2
                        |||
                        STATION_3
                    )
                    |[PRED1, SUCC1, PRED2, SUCC2, PRED3, SUCC3]|
                    (
                        LINK_1
                        |||
                        LINK_2
                        |||
                        LINK_3
                    )
                )
        ==
            "SERVICE_WITHOUT_CRASHES.bcg" ;
% }

(* function invocations *)
% F 01
% F 02
% F 03
% F 04
```

B Comparing an SVL description and a Makefile

This example is based upon a case-study presented in [Pec99], dealing with the verification of a Cluster File System (CFS). It shows how a 2 page long SVL description can replace a 5 page long MAKEFILE performing the same compositional verification, thus illustrating the gain in conciseness and clarity offered by SVL.

B.1 SVL description

```
% DEFAULT_LOTOS_FILE="cfs.lotos"

"site123medium.bcg" = generation of
    leaf branching reduction of
        (
            (
                OutputCell1With23
                |||
                OutputCell2With13
                |||
                OutputCell3With12
            )
            |[ SEND, RCV ]|
            (
                Master1With23
                |||
                Site2With13
                |||
                Site3With12
            )
        );

"cfs123.bcg" = branching reduction of
    generation of hide "SEND", "RCV" in "site123medium.bcg";

% echo
% echo "*****************************************************************"
% echo "Verifying on cfs123.bcg properties defined in cfs.xtl"
% echo "-----------------------------------------------------"
% xtl cfs.xtl cfs123.bcg
% echo "*****************************************************************"

"complete123.bcg" = branching reduction of
    generation of leaf branching reduction of
        (
            "site123medium.bcg"
```

```
            |[ CFSREQ, CFSANS, SEND ]|
            (
                InitMemory
                |[ READ, WRITE ]|
                (
                    GeneralUser1
                    |||
                    GeneralUser2
                    |||
                    GeneralUser3
                )
            )
        );

"abstract123.bcg" = branching reduction of generation of
    hide "CFS...", "SEND", "RCV" in "complete123.bcg";

% echo
% echo "*****************************************************************"
% echo "Verifying on abstract123.bcg properties defined in data.xtl"
% echo "-----------------------------------------------------------"
% xtl data.xtl abstract123.bcg
% echo "*****************************************************************"

"abstract123_12.bcg" = branching reduction of generation of
    (
        (
            total rename "\([A-Z0-9]*\) \(.*\)" -> "\1 !ADDR1 \2" in
                "abstract123.bcg"
        )
        |||
        (
            total rename "\([A-Z0-9]*\) \(.*\)" -> "\1 !ADDR2 \2" in
                "abstract123.bcg"
        )
    );

% echo
% echo "*****************************************************************"
% echo "Verifying on abstract123_12.bcg properties defined in data_12.xtl"
% echo "-----------------------------------------------------------------"
% xtl data_12.xtl abstract123_12.bcg
% echo "*****************************************************************"
```

B.2 Makefile description

```
#! /bin/make

CAESAR          = caesar
CAESAR_ADT      = caesar.adt
MIN             = aldebaran -omin -std
EXPOPEN         = exp.open
CAESAROPEN      = caesar.open
BCGOPEN         = bcg_open
INFO            = aldebaran -info
XTL             = xtl
GENERATOR       = generator

################################################################
# XTL verifications
################################################################

all:    cfs-cfs \
        data-abstract \
        data-abstract-2

cfs-cfs: cfs.xtl cfs123.bcg
        $(XTL) cfs.xtl cfs123.bcg

data-abstract: data.xtl abstract123.bcg
        $(XTL) data.xtl abstract123.bcg

data-abstract-2: data_12.xtl abstract123_12.bcg
        $(XTL) data_12.xtl abstract123_12.bcg

################################################################
# process LOTOS file.
################################################################

cfs.h: cfs.lotos
        $(CAESAR_ADT) cfs.lotos

cfs.bcg: cfs.lotos cfs.h
        $(CAESAR) cfs.lotos

################################################################
# Implicit rules
################################################################

# minimize a LTS
```

```
%_omin.bcg: %.bcg
	$(MIN) -output $@ $<

###############################################################
# Targets for compositional LTS generation
###############################################################

master1with23.bcg: cfs.lotos cfs.h
	$(CAESAR) -root Master1With23 cfs.lotos
	$(MIN) -output $@ cfs.bcg
	$(INFO) $@
	rm -f cfs.bcg

site2with13.bcg: cfs.lotos cfs.h
	$(CAESAR) -root Site2With13 cfs.lotos
	$(MIN) -output $@ cfs.bcg
	$(INFO) $@
	rm -f cfs.bcg

site3with12.bcg: cfs.lotos cfs.h
	$(CAESAR) -root Site3With12 cfs.lotos
	$(MIN) -output $@ cfs.bcg
	$(INFO) $@
	rm -f cfs.bcg

outputcell1with23.bcg: cfs.lotos cfs.h
	$(CAESAR) -root OutputCell1With23 cfs.lotos
	$(MIN) -output $@ cfs.bcg
	$(INFO) $@
	rm -f cfs.bcg

outputcell2with13.bcg: cfs.lotos cfs.h
	$(CAESAR) -root OutputCell2With13 cfs.lotos
	$(MIN) -output $@ cfs.bcg
	$(INFO) $@
	rm -f cfs.bcg

outputcell3with12.bcg: cfs.lotos cfs.h
	$(CAESAR) -root OutputCell3With12 cfs.lotos
	$(MIN) -output $@ cfs.bcg
	$(INFO) $@
	rm -f cfs.bcg

memory.bcg: cfs.lotos cfs.h
	$(CAESAR) -root InitMemory cfs.lotos
	$(MIN) -output $@ cfs.bcg
```

```
	$(INFO) $@
	rm -f cfs.bcg

user1.bcg: cfs.lotos cfs.h
	$(CAESAR) -root GeneralUser1 cfs.lotos
	$(MIN) -output $@ cfs.bcg
	$(INFO) $@
	rm -f cfs.bcg

user2.bcg: cfs.lotos cfs.h
	$(CAESAR) -root GeneralUser2 cfs.lotos
	$(MIN) -output $@ cfs.bcg
	$(INFO) $@
	rm -f cfs.bcg

user3.bcg: cfs.lotos cfs.h
	$(CAESAR) -root GeneralUser3 cfs.lotos
	$(MIN) -output $@ cfs.bcg
	$(INFO) $@
	rm -f cfs.bcg

################################################################
# compositional generation and minimization
################################################################

medium123output.bcg: outputcell1with23.bcg outputcell2with13.bcg \
		outputcell3with12.bcg medium123output.exp
	$(EXPOPEN) medium123output.exp $(GENERATOR) TMP.bcg
	$(MIN) -output $@ TMP.bcg
	$(INFO) $@
	rm -f TMP.bcg

user123memory.bcg: user1.bcg user2.bcg user3.bcg memory.bcg \
		user123memory.exp
	$(EXPOPEN) user123memory.exp $(GENERATOR) TMP.bcg
	$(MIN) -output $@ TMP.bcg
	$(INFO) $@
	rm -f TMP.bcg

site123medium.bcg: master1with23.bcg site2with13.bcg site3with12.bcg \
		medium123output.bcg site123medium.exp
	$(EXPOPEN) site123medium.exp $(GENERATOR) TMP.bcg
	$(MIN) -output $@ TMP.bcg
	$(INFO) $@
	rm -f TMP.bcg
```

```
cfs123.bcg: site123medium.bcg sendrcv.hide
        $(MIN) -output $@ -hide sendrcv.hide site123medium.bcg
        $(INFO) $@

complete123.bcg: user123memory.bcg site123medium.bcg complete123.exp
        $(EXPOPEN) complete123.exp $(GENERATOR) TMP.bcg
        $(MIN) -output $@ TMP.bcg
        $(INFO) $@
        rm -f TMP.bcg

abstract123.bcg: complete123.bcg cfssendrcv.hide
        $(MIN) -output $@ -hide cfssendrcv.hide complete123.bcg
        $(INFO) $@

abstract123_12.bcg: abstract123_1.bcg abstract123_2.bcg abstract123_12.exp
        $(EXPOPEN) abstract123_12.exp $(GENERATOR) TMP.bcg
        $(MIN) -output $@ TMP.bcg
        $(INFO) $@
        rm -f TMP.bcg

################################################################
# Generation of isomorphic graphs by renaming using bcg_labels
################################################################

# add first attribute "!ADDR1" to all transitions

abstract123_1.rename:
        @echo 'rename' > $@
        @echo '"\([A-Z0-9]*\) \(.*\)" -> "\1 !ADDR1 \2"' >> $@

abstract123_1.bcg: abstract123_1.rename abstract123.bcg
        bcg_labels -rename abstract123_1.rename abstract123.bcg $@

# add first attribute "!ADDR2" to all transitions

abstract123_2.rename:
        @echo 'rename' > $@
        @echo '"\([A-Z0-9]*\) \(.*\)" -> "\1 !ADDR2 \2"' >> $@

abstract123_2.bcg: abstract123_2.rename abstract123.bcg
        bcg_labels -rename abstract123_2.rename abstract123.bcg $@

################################################################
# Auxiliary text files, re-generated on demand.
################################################################
```

```
medium123output.exp:
        echo '       outputcell1with23' > $@
        echo '   ||| outputcell2with13' >> $@
        echo '   ||| outputcell3with12' >> $@

site123medium.exp:
        echo 'medium123output |[SEND,RCV]|' > $@
        echo '(master1with23 ||| site2with13 ||| site3with12)' >> $@

user123memory.exp:
        echo 'memory |[READ,WRITE]| (user1 ||| user2 ||| user3)' > $@

complete123.exp:
        echo 'site123medium |[CFSREQ,CFSANS,SEND]| user123memory' > $@

abstract123_12.exp:
        echo 'abstract123_1 ||| abstract123_2' > $@

sendrcv.sync:
        echo 'Sync' > $@
        echo 'SEND' >> $@
        echo 'RCV' >> $@

sendrcv.hide:
        echo 'hide' > $@
        echo 'SEND .*' >> $@
        echo 'RCV .*' >> $@

cfssendrcv.hide:
        echo 'hide' > $@
        echo 'CFS... .*' >> $@
        echo 'SEND .*' >> $@
        echo 'RCV .*' >> $@
```

References

[BD97] Pierre Boullier and Philippe Deschamp. Le système SYNTAX : Manuel d'utilisation et de mise en œuvre sous Unix. `http://www-rocq.inria.fr/oscar/www/syntax`, October 1997.

[BK84] J. A. Bergstra and J. W. Klop. Process Algebra for Synchronous Communication. *Information and Computation*, 60:109–137, 1984.

[BM91] Simon Bainbridge and Laurent Mounier. Specification and Verification of a Reliable Multicast Protocol. Technical Report HPL-91-163, Hewlett-Packard Laboratories, Bristol, U.K., October 1991.

[BRRd96] Amar Bouali, Annie Ressouche, Valérie Roy, and Robert de Simone. The Fc2Tools set: a Toolset for the Verification of Concurrent Systems. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA)*, volume 1102 of *Lecture Notes in Computer Science*. Springer Verlag, August 1996.

[CGM+96] Ghassan Chehaibar, Hubert Garavel, Laurent Mounier, Nadia Tawbi, and Ferruccio Zulian. Specification and Verification of the PowerScale Bus Arbitration Protocol: An Industrial Experiment with LOTOS. In Reinhard Gotzhein and Jan Bredereke, editors, *Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification FORTE/PSTV'96 (Kaiserslautern, Germany)*, pages 435–450. IFIP, Chapman & Hall, October 1996. Full version available as INRIA Research Report RR-2958.

[CGMdP01] Manuel Aguilar Cornejo, Hubert Garavel, Radu Mateescu, and Noël de Palma. Specification and Verification of a Dynamic Reconfiguration Protocol for Agent-Based Applications. In Aleksander Laurentowski, Jacek Kosinski, Zofia Mossurska, and Radoslaw Ruchala, editors, *Proceedings of the 3rd IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems DAIS'2001 (Krakow, Poland)*. IFIP, Kluwer Academic Publishers, September 2001.

[Che98] K. H. Cheung. *Compositional Analysis of Complex Distributed Systems*. PhD thesis, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong, 1998.

[CK93] S. C. Cheung and J. Kramer. Enhancing Compositional Reachability Analysis with Context Constraints. In *Proceedings of the 1st ACM SIGSOFT International Symposium on the Foundations of Software Engineering (Los Angeles, CA, USA)*, pages 115–125. ACM Press, December 1993.

[CK95] S. C. Cheung and J. Kramer. Compositional Reachability Analysis of Finite-State Distributed Systems with User-Specified Constraints. In *Proceedings of the 3rd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (Washington, DC, USA)*, pages 140–150. ACM Press, October 1995.

[CK96] S. C. Cheung and J. Kramer. Context Constraints for Compositional Reachability. *ACM Transactions on Software Engineering Methodology TOSEM*, 5(4):334–377, October 1996.

[Fer88] Jean-Claude Fernandez. *ALDEBARAN : un système de vérification par réduction de processus communicants.* Thèse de Doctorat, Université Joseph Fourier (Grenoble), May 1988.

[FGK+96] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Radu Mateescu, Laurent Mounier, and Mihaela Sighireanu. CADP (CÆSAR/ALDEBARAN Development Package): A Protocol Validation and Verification Toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA)*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer Verlag, August 1996.

[FGM+92] Jean-Claude Fernandez, Hubert Garavel, Laurent Mounier, Anne Rasse, Carlos Rodríguez, and Joseph Sifakis. A Toolbox for the Verification of LOTOS Programs. In Lori A. Clarke, editor, *Proceedings of the 14th International Conference on Software Engineering ICSE'14 (Melbourne, Australia)*, pages 246–259. ACM, May 1992.

[Gar89] Hubert Garavel. *Compilation et vérification de programmes LOTOS.* Thèse de Doctorat, Université Joseph Fourier (Grenoble), November 1989.

[Gar96] Hubert Garavel. An Overview of the Eucalyptus Toolbox. In Z. Brezočnik and T. Kapus, editors, *Proceedings of the COST 247 International Workshop on Applied Formal Methods in System Design (Maribor, Slovenia)*, pages 76–88. University of Maribor, Slovenia, June 1996.

[Gar98] Hubert Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In Bernhard Steffen, editor, *Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98 (Lisbon, Portugal)*, volume 1384 of *Lecture Notes in Computer Science*, pages 68–84, Berlin, March 1998. Springer Verlag. Full version available as INRIA Research Report RR-3352.

[Gia99] D. Giannakopoulou. *Model Checking for Concurrent Software Architectures.* PhD thesis, Imperial College of Science, Technology and Medicine — University of London — Department of Computer Science, January 1999.

[GM97] Hubert Garavel and Laurent Mounier. Specification and Verification of Various Distributed Leader Election Algorithms for Unidirectional Ring Networks. *Science of Computer Programming*, 29(1–2):171–197, July 1997. Special issue on Industrially Relevant Applications of Formal Analysis Techniques. Full version available as INRIA Research Report RR-2986.

[GS90] Susanne Graf and Bernhard Steffen. Compositional Minimization of Finite State Systems. In R. P. Kurshan and E. M. Clarke, editors, *Proceedings of the 2nd Workshop on Computer-Aided Verification (Rutgers, New Jersey, USA)*, volume 531 of *Lecture Notes in Computer Science*, pages 186–196. Springer Verlag, June 1990.

[GS98] Hubert Garavel and Mihaela Sighireanu. Towards a Second Generation of Formal Description Techniques – Rationale for the Design of E-LOTOS. In Jan-Friso Groote, Bas Luttik, and Jos van Wamel, editors, *Proceedings of the 3rd International Workshop on Formal Methods for Industrial Critical Systems FMICS'98 (Amsterdam, The Netherlands)*, pages 187–230, Amsterdam, May 1998. CWI. Invited lecture.

[GS99] Hubert Garavel and Mihaela Sighireanu. A Graphical Parallel Composition Operator for Process Algebras. In Jianping Wu, Qiang Gao, and Samuel T. Chanson, editors, *Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification FORTE/PSTV'99 (Beijing, China)*, pages 185–202. IFIP, Kluwer Academic Publishers, October 1999.

[GSL96] S. Graf, B. Steffen, and G. Lüttgen. Compositional Minimisation of Finite State Systems using Interface Specifications. *Formal Aspects of Computation*, 8, September 1996.

[GVZ01] Hubert Garavel, César Viho, and Massimo Zendri. System Design of a CC-NUMA Multiprocessor Architecture using Formal Specification, Model-Checking, Co-Simulation, and Test Generation. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 3(3), July 2001. Also available as INRIA Research Report RR-4041.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[ISO88] ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1988.

[KM97] Jean-Pierre Krimm and Laurent Mounier. Compositional State Space Generation from LOTOS Programs. In Ed Brinksma, editor, *Proceedings of TACAS'97*

Tools and Algorithms for the Construction and Analysis of Systems (University of Twente, Enschede, The Netherlands), volume 1217 of *Lecture Notes in Computer Science*, Berlin, April 1997. Springer Verlag. Extended version with proofs available as Research Report VERIMAG RR97-01.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[MS00] Radu Mateescu and Mihaela Sighireanu. Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. In Stefania Gnesi, Ina Schieferdecker, and Axel Rennoch, editors, *Proceedings of the 5th International Workshop on Formal Methods for Industrial Critical Systems FMICS'2000 (Berlin, Germany)*, GMD Report 91, pages 65–86, Berlin, April 2000. Also available as INRIA Research Report RR-3899.

[MSGS88] J. Malhotra, S. A. Smolka, A. Giacalone, and R. Shapiro. A Tool for Hierarchical Design and Simulation of Concurrent Systems. In *Proceedings of the BCS-FACS Workshop on Specification and Verification of Concurrent Systems (Stirling, Scotland)*, pages 140–152, Swinton, UK, July 1988. British Computer Society.

[Pec99] Charles Pecheur. Advanced Modelling and Verification Techniques Applied to a Cluster File System. In Robert J. Hall and Ernst Tyugu, editors, *Proceedings of the 14th IEEE International Conference on Automated Software Engineering ASE-99 (Cocoa Beach, Florida, USA).* IEEE Computer Society, October 1999. Extended version available as INRIA Research Report RR-3416.

[Que01] Juan Quemada, editor. Information Technology – Enhancements to LOTOS (E-LOTOS). ISO/IEC FDIS 15437 ballot, May 2001.

[SBC+00] Mihaela Sighireanu, Xavier Bouchoux, Claude Chaudet, Hubert Garavel, Marc Herbert, Frédéric Lang, and Bruno Vivien. TRAIAN: A Compiler for E-LOTOS/LOTOS NT Specifications. `http://www.inrialpes.fr/vasy/traian/`, November 2000.

[Sig00] 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.

[SLU89] K. K. Sabnani, A. M. Lapone, and M. U. Uyar. An Algorithmic Procedure for Checking Safety Properties of Protocols. *IEEE Transactions on Communications*, 37(9):940–948, September 1989.

[TK93a] K. C. Tai and V. Koppol. Hierarchy-Based Incremental Reachability Analysis of Communication Protocols. In *Proceedings of the IEEE International Conference on Network Protocols (San Francisco, CA)*, pages 318–325, Piscataway, NJ, October 1993. IEEE Press.

[TK93b] K. C. Tai and V. Koppol. An Incremental Approach to Reachability Analysis of Distributed Programs. In *Proceedings of the 7th International Workshop on Software Specification and Design (Los Angeles, CA)*, pages 141–150, Piscataway, NJ, December 1993. IEEE Press.

[Val93] Antti Valmari. Compositional State Space Generation. In *Proceedings of Advances in Petri Nets*, volume 674 of *Lecture Notes in Computer Science*, pages 427–457. Springer Verlag, 1993.

[Yeh93] W. J. Yeh. *Controlling State Explosion in Reachability Analysis*. PhD thesis, Software Engineering Research Center (SERC) Laboratory, Purdue University, December 1993. Technical Report SERC-TR-147-P.

[YY91] W. J. Yeh and M. Young. Compositional Reachability Analysis Using Process Algebra. In *Proceedings of the ACM SIGSOFT Symposium on Testing, Analysis, and Verification (SIGSOFT'91, Victoria, British Columbia, Canada)*, pages 49–59, New York, NY, October 1991. ACM Press.

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

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité 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