SVL: A SCRIPTING LANGUAGE FOR COMPOSITIONAL VERIFICATION

Hubert Garavel, Frédéric Lang*

INRIA Rhône-Alpes, 655, avenue de l'Europe, F-38330 Montbonnot St. Martin, France

Hubert.Garavel@inria.fr, Frederic.Lang@inria.fr

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 article 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.

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

*This work has been partially supported by DYADE, the BULL/INRIA joint venture for advanced research

symbolic verification, on-the-fly verification, partial orders, symmetries, data flow analysis, and compositional verification.

This article 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 labeled transition systems (LTSS, for short). The sequential processes are composed in parallel, either in a flat or hierarchical manner.

In its simplest form [9, 23, 27, 32, 28, 29, 30], 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 [32] for a discussion on this issue).

Although this simple form of compositional verification has been applied successfully to some complex systems (e.g., [11, 4] in the case of the LOTOS language [20]), 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 [15, 6, 31, 7, 8, 18, 22, 5, 14], 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 [14] and the PROJECTOR/DES2AUT tools [22]. The latter tools are part of the CADP protocol engineering toolbox [10] and have been applied to in-

dustrial case-studies [22, 24, 19]. 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 article is organized as follows. Section 2 gives a few preliminary definitions. Section 3 briefly presents the principles and limitations of the DES2AUT tool of [22]. 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. Finally, Section 7 gives concluding remarks and lists directions for future work.

2. DEFINITIONS

Labelled Transition Systems are the natural model for action-based specification languages, especially process algebras such as LOTOS [20]. 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 [3], 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 $\text{succ} : S \rightarrow 2^T$ defined by $\text{succ}(s) = \{(s, a, s') \mid (s, a, s') \in T\}$. A generic representation of implicit LTSS is provided by the language-independent environment OPEN/CÆSAR [13] embedded in CADP. OPEN/CÆSAR offers primitives for accessing the initial state of an LTS and for enumer-

ating 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	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 [22]. 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 $F, P, G_1, \dots, G_n$ are terminal symbols denoting respectively file prefixes, LOTOS process identifiers, and LOTOS gate identifiers:

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

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 $G_1, \dots, G_n$);
- (2) the behavior B_0 in which gates $G_1, \dots, G_n$ are hidden using the LOTOS hiding operator;
 - (3) the behaviors B_1 and B_2 executing in parallel with synchronization and communication on gates $G_1, \dots, G_n$ according to the LOTOS parallel composition semantics;
 - (4) the behavior B_1 restricted by synchronizing it on the set of gates $G_1, \dots, G_n$ 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 DE S2AUT tool has been established on significant case-studies [22, 24, 19], 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,

$$(\text{User1.aut} \mid \mid \mid \text{User2.aut}) \mid [G] \mid \text{Medium.aut}$$

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.
- 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, $G, G_1, \dots, G_n$ denote LOTOS gate identifiers, $L, L_1, \dots, L_n$ 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:

```

 $op$  ::= <= | == | >=
 $par$  ::= |[ $G_1, \dots, G_n$ ]| | ||| | ||
 $E$  ::= aut | bcg | exp | fc2 | lotos | seq
 $R$  ::= branching | strong | observational | safety | tau*.a | ...
 $M$  ::= std | fly | bdd
 $T$  ::= aldebaran | bcg_min | evaluator | exhibitor | fc2tools | ...
 $U$  ::= user | <empty>

```

where op denotes an equivalence or preorder relation, par a LOTOS parallel operator, 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 " $S_1; \dots; S_n$ ".

$S ::=$	" $F.E$ " = B_0	(S1)
	" $F.E$ " = R comparison [using M] [with T] B_1 <i>op</i> B_2	(S2)
	" $F.E$ " = deadlock [with T] of B_0	(S3)
	" $F.E$ " = livelock [with T] of B_0	(S4)
$B ::=$	" $F.E$ " " $F.lotos$ " : $P[G_1, \dots, G_n]$	(B1)
	hide $L_1, \dots, L_n$ in B_0	(B2)
	B_1 <i>par</i> B_2	(B3)
	rename $L_1 \rightarrow L'_1, \dots, L_n \rightarrow L'_n$ in B_0	(B4)
	generation of B_0	(B5)
	R reduction [using M] [with T] of B_0	(B6)
	U abstraction B_1 sync $G_1, \dots, G_n$ of B_2	(B7)

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 $\llbracket B \rrbracket_\sigma$, 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 $\llbracket B \rrbracket_\sigma$ 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 $pref(\sigma)$ 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 $\llbracket B \rrbracket_\sigma$ returns B else it converts the LTS contained in B to $pref(\sigma)$ 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 **ALDEBARAN**, **BCG_LABELS**, and **EXP.OPEN** tools. This generalizes the LOTOS hiding operator by allowing regular expressions on

labels. 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 $L_1, \dots, L_n$ are LOTOS gates and B_0 is a parallel composition of behaviors, then $\llbracket B \rrbracket_\sigma$ returns the EXP composition expression **hide** $L_1, \dots, L_n$ **in** $\llbracket B_0 \rrbracket_{\{\mathbf{exp}, \mathbf{aut}, \mathbf{bcg}, \mathbf{fc2}, \mathbf{seq}\}}$, else $\llbracket B \rrbracket_\sigma$ returns the conversion to $\mathit{pref}(\sigma)$ of $\llbracket B_0 \rrbracket_{\{\mathbf{bcg}\}}$ in which labels matching one of $L_1, \dots, L_n$ are hidden.

- (B3) denotes the parallel composition of B_1 and B_2 . σ must contain **exp** or **fc2**. $\llbracket B \rrbracket_\sigma$ returns the EXP composition expression $\llbracket B_1 \rrbracket_{\sigma'} \text{ par } \llbracket B_2 \rrbracket_{\sigma'}$ where $\sigma' = \{\mathbf{exp}, \mathbf{aut}, \mathbf{bcg}, \mathbf{fc2}, \mathbf{seq}\}$.
- (B4) denotes the label renaming of an explicit LTS using the BCG_LABELS tool, which supports UNIX-like regular expression matching and substring replacement. $\llbracket B \rrbracket_\sigma$ returns the conversion to $\mathit{pref}(\sigma)$ of $\llbracket B_0 \rrbracket_{\{\mathbf{bcg}\}}$ in which labels are renamed as specified by the rules $L_1 \rightarrow L'_1, \dots, L_n \rightarrow L'_n$.
- (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 $\llbracket B_0 \rrbracket_{\sigma \cup \{\mathbf{exp}, \mathbf{fc2}, \mathbf{lotos}\}}$. If α is already an explicit LTS, then $\llbracket B \rrbracket_\sigma$ returns α else it returns the conversion of α to an explicit LTS of format $\mathit{pref}(\sigma)$.
- (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*. $\llbracket B \rrbracket_\sigma$ returns the conversion to format $\mathit{pref}(\sigma)$ of the *RMT-reduction* of $\llbracket B_0 \rrbracket_{\sigma^T}$, where σ^T is the set of the accepted input formats for reduction with tool T .
- (B7) denotes the abstraction w.r.t. interface constraints, which is computed using OPEN/CÆSAR compilers and PROJECTOR. $\llbracket B \rrbracket_\sigma$ returns the conversion to format $\mathit{pref}(\sigma)$ of the abstraction of $\llbracket B_2 \rrbracket_{\{\mathbf{bcg}, \mathbf{exp}, \mathbf{fc2}, \mathbf{lotos}\}}$ w.r.t. the interface $\llbracket B_1 \rrbracket_{\{\mathbf{aut}\}}$. The “**user**” keyword has the same meaning as the “?” symbol of the DES Language.

The statements have the following effects:

- (S1) stores in file $F.E$ (where $E \neq \mathbf{lotos}$) either $\llbracket B_0 \rrbracket_{\{E\}}$ if E denotes an explicit LTS format or $\llbracket B_0 \rrbracket_{\{\mathbf{aut}, \mathbf{bcg}, \mathbf{exp}, \mathbf{fc2}, \mathbf{seq}\}}$ 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 $\llbracket B_1 \rrbracket_{\sigma^T}$ and $\llbracket B_2 \rrbracket_{\sigma^T}$, where σ^T denotes the set of possible input formats accepted by T . The result stored in file $F.E$ (where $E \notin \{\mathbf{exp}, \mathbf{lotos}\}$) is a (set of) distinguishing path(s) if the comparison returns false or an empty path otherwise.
- (S3) detects deadlocks using a tool T (ALDÉBARAN, EVALUATOR, EXHIBITOR, or FC2TOOLS) in $\llbracket B_0 \rrbracket_{\sigma^T}$, where σ^T is the set of possible input formats accepted by T . The result stored in file $F.E$ (where $E \notin \{\mathbf{exp}, \mathbf{lotos}\}$) is a (set of) path(s) leading to (one or all) deadlock state(s), if any, or an empty path otherwise.
- (S4) checks for livelocks in a way similar to statement (S3).

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:

$$B ::= \dots$$

$$| \quad A \text{ R reduction using } M \text{ with } T \text{ of } B_0$$

where the attribute A is defined by:

$$A ::= \mathbf{leaf} \mid \mathbf{root\ leaf} \mid \mathbf{node}$$

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 $\varepsilon_{RMT}(B, A)$ where ε_{RMT} is defined as follows and where the shorthand notation $R_{RMT}(B)$ stands for an RMT -reduction of B :

```

 $\varepsilon_{RMT}(B, A) =$  if  $A =$  “root leaf” then  $\mathcal{R}_{RMT}(\varepsilon_{RMT}(B, \text{leaf}))$ 
else case  $B$  in
  rename ... in  $B_0 \rightarrow \mathcal{R}_{RMT}(\text{rename ... in } \varepsilon_{RMT}(B_0, A))$ 
  |  $U$  abstraction  $B_1 \dots$  of  $B_2 \rightarrow$ 
     $\mathcal{R}_{RMT}(U \text{ abstraction } B_1 \dots \text{ of } \varepsilon_{RMT}(B_2, A))$ 
  |  $B_1 \text{ par } B_2 \rightarrow$ 
    if  $A =$  “node” then  $\mathcal{R}_{RMT}(\varepsilon_{RMT}(B_1, A) \text{ par } \varepsilon_{RMT}(B_2, A))$ 
    else  $\varepsilon_{RMT}(B_1, A) \text{ par } \varepsilon_{RMT}(B_2, A)$ 
  | hide  $L_1, \dots, L_n$  in  $B_0 \rightarrow$  if  $\varepsilon_{RMT}(B_0, A)$  has the form “ $B_1 \text{ par } B_2$ ” and
    all  $L_i$  are LOTOS gates then hide  $L_1, \dots, L_n$  in  $\varepsilon_{RMT}(B_0, A)$ 
    else  $\mathcal{R}_{RMT}(\text{hide } L_1, \dots, L_n \text{ in } \varepsilon_{RMT}(B_0, A))$ 
  | 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 ε_{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 $R_{RMT}(\mathcal{R}_{R'M'T'}(B))$ 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 .

```

 $\mathcal{O}(B) =$  case  $B$  in
  “ $F.E''[:P[\dots]] \rightarrow B$ ”
  |  $B_1 \text{ par } B_2 \rightarrow \mathcal{O}(B_1) \text{ par } \mathcal{O}(B_2)$ 
  | generation of  $B_0 \rightarrow$  generation of  $\mathcal{O}(B_0)$ 
  | rename ... in  $B_0 \rightarrow$  rename ... in  $\mathcal{O}(B_0)$ 
  |  $U$  abstraction  $B_1 \dots$  of  $B_2 \rightarrow U$  abstraction  $\mathcal{O}(B_1) \dots$  of  $\mathcal{O}(B_2)$ 
  | hide  $L_1, \dots, L_n$  in  $B_0 \rightarrow$ 
    if  $B_0$  has the form “hide  $L_{n+1}, \dots, L_{n+m}$  in  $B'_0$ ” then
       $\mathcal{O}(\text{hide } L_1, \dots, L_{n+m} \text{ in } B'_0)$ 
    else hide  $L_1, \dots, L_n$  in  $\mathcal{O}(B_0)$ 
  |  $\mathcal{R}_{RMT}(B_0) \rightarrow$  case  $B_0$  in
     $\mathcal{R}_{R'M'T'}(B'_0)$  where  $R' \sqsubseteq R \rightarrow \mathcal{O}(\mathcal{R}_{R'M'T'}(B'_0))$ 
    | hide  $L_1, \dots, L_n$  in  $\mathcal{R}_{R'M'T'}(B'_0)$  where  $R = R' \rightarrow$ 
       $\mathcal{O}(\mathcal{R}_{RMT}(\text{hide } L_1, \dots, L_n \text{ in } B'_0))$ 
    |  $U$  abstraction  $B_1 \dots$  of  $\mathcal{R}_{R'M'T'}(B_2)$  where  $R = R' \rightarrow$ 
       $\mathcal{O}(\mathcal{R}_{RMT}(U \text{ abstraction } B_1 \dots \text{ of } B_2))$ 
    | otherwise  $\rightarrow \mathcal{R}_{RMT}(\mathcal{O}(B_0))$ 

```

Figure 1. Architecture of SVL 2.0

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 1. 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 [2]. 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 LOTOS NT language [16, 26]. Inspired by the standardization work on Enhanced LOTOS [21], 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 [25] 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 [4, 22, 24, 19]. 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 [12] 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. Switch-

ing 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 [1]. 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:

$$\text{"a.bcg"} = \text{leaf strong reduction of} \\ ((B_1 \mid [G] \mid B_0) \parallel (B_2 \mid [G] \mid B_0))$$

can be evaluated more efficiently as:

$$\begin{aligned} \text{"b.bcg"} &= \text{strong reduction of } B_0 \\ \text{"a.bcg"} &= ((\text{strong reduction of } B_1) \mid [G] \mid \text{"b.bcg"}) \parallel \\ &\quad ((\text{strong reduction of } B_2) \mid [G] \mid \text{"b.bcg"}) \end{aligned}$$

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 [17] 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 paper.

REFERENCES

- [1] M. Aguilar Cornejo, H. Garavel, R. Mateescu, N. de Palma. Specification and Verification of a Dynamic Reconfiguration Protocol for Agent-Based Applications. In *Proc. of DAIS'01*. Kluwer Academic Publishers.
- [2] P. Boullier, P. Deschamp. Le système SYNTAX : Manuel d'utilisation et de mise en œuvre sous Unix. <http://www-rocq.inria.fr/oscar/www/syntax>, 1997.
- [3] A. Bouali, A. Ressouche, V. Roy, and R. de Simone. The Fc2Tools set: a Toolset for the Verification of Concurrent Systems. In *Proc. of CAV'96*, LNCS vol. 1102.
- [4] G. Chehaibar, H. Garavel, L. Mounier, N. Tawbi, and F. Zulian. Specification and Verification of the PowerScale Bus Arbitration Protocol: An Industrial Experiment with LOTOS. In *Proc. of FORTE/PSTV'96*. IFIP.
- [5] K. H. Cheung. *Compositional Analysis of Complex Distributed Systems*. Ph.D. thesis, Hong Kong Univ. of Science and Technology, 1998.
- [6] S. C. Cheung and J. Kramer. Enhancing Compositional Reachability Analysis with Context Constraints. In *Proc. of FSE'93*. ACM.
- [7] S. C. Cheung and J. Kramer. Compositional Reachability Analysis of Finite-State Distributed Systems with User-Specified Constraints. In *Proc. of FSE'95*. ACM.
- [8] S. C. Cheung and J. Kramer. Context Constraints for Compositional Reachability. *ACM Transactions on Software Engineering Methodology*, 1996.
- [9] J.-C. Fernandez. *ALDEBARAN : un système de vérification par réduction de processus communicants*. Thèse de Doctorat, Univ. Joseph Fourier (Grenoble), 1988.
- [10] J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP (CÆSAR/ALDEBARAN Development Package): A Protocol Validation and Verification Toolbox. In *Proc. of CAV'96*, LNCS vol. 1102.
- [11] J.-C. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodríguez, and J. Sifakis. A Toolbox for the Verification of LOTOS Programs. In *Proc. of ICSE'92*. ACM.
- [12] H. Garavel. An Overview of the Eucalyptus Toolbox. In *Proc. of the Int. Workshop on Applied Formal Methods in System Design*. Univ. of Maribor, Slovenia, 1996.
- [13] H. Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In *Proc. of TACAS'98*. LNCS vol. 1384.
- [14] D. Giannakopoulou. *Model Checking for Concurrent Software Architectures*. Ph.D. thesis, Imp. College of Science, Technology and Medicine — Univ. of

London, 1999.

- [15] S. Graf and B. Steffen. Compositional Minimization of Finite State Systems. In *Proc. of CAV'90*, LNCS vol. 531.
- [16] H. Garavel and M. Sighireanu. Towards a Second Generation of Formal Description Techniques – Rationale for the Design of E-LOTOS. In *Proc. of FMICS'98*. CWI.
- [17] H. Garavel and M. Sighireanu. A Graphical Parallel Composition Operator for Process Algebras. In *Proc. of FORTE/PSTV'99*. IFIP.
- [18] S. Graf, B. Steffen, and G. Lüttgen. Compositional Minimisation of Finite State Systems using Interface Specifications. *Formal Aspects of Computation*, Sept. 1996.
- [19] H. Garavel, C. Viho, and M. Zendri. System Design of a CC-NUMA Multiprocessor Architecture using Formal Specification, Model-Checking, Co-Simulation, and Test Generation. *STTT*. Springer-Verlag, July 2001.
- [20] ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, ISO — Information Processing Systems — OSI, September 1988.
- [21] ISO/IEC. Information Technology – Enhancements to LOTOS (E-LOTOS). Juan Quemada, editor, FDIS 15437 ballot, May 2001.
- [22] J.-P. Krimm and L. Mounier. Compositional State Space Generation from LOTOS Programs. In *Proc. of TACAS'97*. LNCS vol. 1217.
- [23] J. Malhotra, S. A. Smolka, A. Giacalone, and R. Shapiro. A Tool for Hierarchical Design and Simulation of Concurrent Systems. In *Proc. of the Workshop on Specification and Verification of Concurrent Systems*. British Computer Society, 1988.
- [24] C. Pecheur. Advanced Modelling and Verification Techniques Applied to a Cluster File System. In *Proc. of ASE'99*. IEEE Computer Society.
- [25] M. Sighireanu, X. Bouchoux, C. Chaudet, H. Garavel, M. Herbert, F. Lang, and B. Vivien. TRAIAN: A Compiler for E-LOTOS/LOTOS NT Specifications. <http://www.inrialpes.fr/vasy/traian>, 2000.
- [26] M. Sighireanu. LOTOS NT User's Manual (Version 2.1). INRIA projet VASY. <ftp://ftp.inrialpes.fr/pub/vasy/traian/manual.ps.Z>, 2000.
- [27] K. K. Sabnani, A. M. Lapone, and M. U. Uyar. An Algorithmic Procedure for Checking Safety Properties of Protocols. *IEEE Transactions on Communications*, Sept. 1989.
- [28] K. C. Tai and V. Koppol. Hierarchy-Based Incremental Reachability Analysis of Communication Protocols. In *Proc. of the Int. Conf. on Network Protocols*. IEEE, 1993.
- [29] K. C. Tai and V. Koppol. An Incremental Approach to Reachability Analysis of Distributed Programs. In *Proc. of the Int. Workshop on Software Specification and Design*. IEEE, 1993.
- [30] A. Valmari. Compositional State Space Generation. In *Proc. of Advances in Petri Nets*. LNCS vol. 674, 1993.
- [31] W. J. Yeh. *Controlling State Explosion in Reachability Analysis*. Ph.D. thesis, Software Engineering Research Center (SERC) Laboratory, Purdue Univ., Dec. 1993.
- [32] W. J. Yeh and M. Young. Compositional Reachability Analysis Using Process Algebra. In *Proc. of SIGSOFT'91*. ACM.