Compositional Verification using SVL Scripts

Frédéric Lang

Inria Rhône-Alpes - Vasy
655, avenue de l'Europe - F-38330 Montbonnot, France
Frederic.Lang@inria.fr

1 Introduction

User-friendliness of complex software has traditionally been enhanced in two complementary ways: graphical user interfaces and scripting languages. The Cadp toolbox[1] [3, 5] is a complex software suite integrating numerous verification tools. Since 1995, it has been equipped with Eucalyptus, a graphical user interface. However, a dedicated scripting language to automate repetitive verification tasks was still lacking, resulting in ad hoc shell scripts and Makefiles used for this purpose. The main problem was that they were usually too verbose and lacked built-in features to support model-based verification. This has motivated the definition and implementation of the scripting language Svl[2] [4].

An Svl script is a sequence of *statements*, which describe verification operations (such as comparison modulo various equivalence relations, deadlock and livelock detection, verification of temporal logic formulas, etc.) performed on *behaviors*. Basic behaviors are either Labeled Transition Systems (Ltss) described in a number of formats, networks of communicating Ltss, Lotos descriptions, or particular processes in Lotos descriptions. Behaviors can be combined using operations such as parallel composition, label hiding, label renaming, Lts generation, minimization, and abstraction w.r.t. an interface. Svl has also *meta-operations* implementing higher-order strategies for compositional verification.

To execute Svl scripts, a compiler (7,000 lines) has been developed. As depicted in Figure 1, it translates an Svl script into an executable Bourne shell script, which is run to perform the requested operations by calling either the Cadp or the Fc2 [1] tools (e.g., Aldébaran, Bcg_Min, or Fc2Min for minimization). Svl is particularly useful in compositional verification, which we illustrate in this paper with two unpublished examples.

2 Basic Compositional Verification

Compositional verification intends to avoid state explosion by using divide-and-conquer techniques. When verifying a network of concurrent processes, it consists in replacing each process by an *abstraction* (e.g., a minimization modulo an appropriate equivalence relation) simpler than the original process but still preserving the properties to be verified on the whole system.

[1] Cadp web site: "http://www.inrialpes.fr/vasy/cadp".
[2] Svl on-line user-manual: "http://www.inrialpes.fr/vasy/cadp/man/svl.html".

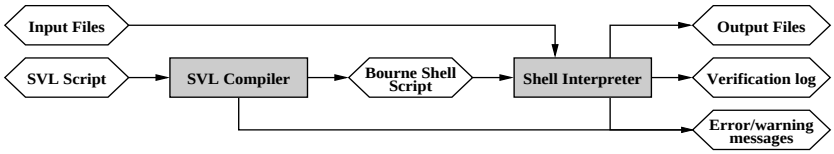

Fig. 1. The SVL tool

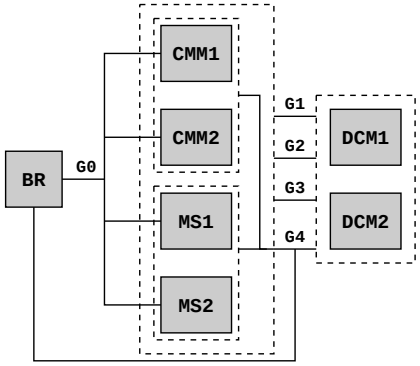

Fig. 2. Architecture of the HAVi Protocol

We illustrate compositional verification with a case study [8] concerning the leader election protocol used in the HAVi standard for home audio/video networks. Figure 2 depicts this protocol modelled in file "`HAVi.lotos`" as a network of seven concurrent processes (`BR`, `DCM1`, etc.) communicating on gates `G0` to `G4`. Due to its complexity, the state space cannot be generated directly, but can be generated compositionally using the following SVL script, which replaces the 85-line MAKEFILE developed by Judi Romijn for the same task:

```
% DEFAULT_LOTOS_FILE="HAVi.lotos"
"HAVi.exp" = leaf strong reduction of                              (* 1 *)
   (BR |[G0, G4]|
    ((DCM1 ||| DCM2)
     |[G1, G2, G3, G4]|
     ((CMM1 |[G0]| CMM2) |[G0, G4]| (MS1 |[G0]| MS2))));
"HAVi.bcg" = strong reduction of "HAVi.exp";                       (* 2 *)
```

In step (1), the LTSs of the seven processes are generated and minimized for strong bisimulation (as specified by the "`leaf reduction`" meta-operation), then composed in parallel to form a network of LTSs named "`HAVi.exp`". In step (2), the LTS corresponding to "`HAVi.exp`" is generated, minimized for strong bisimulation, and stored in file "`HAVi.bcg`" (5,107 states and 18,725 transitions). The verification takes 5 minutes on a standard 450 MHz Linux PC.

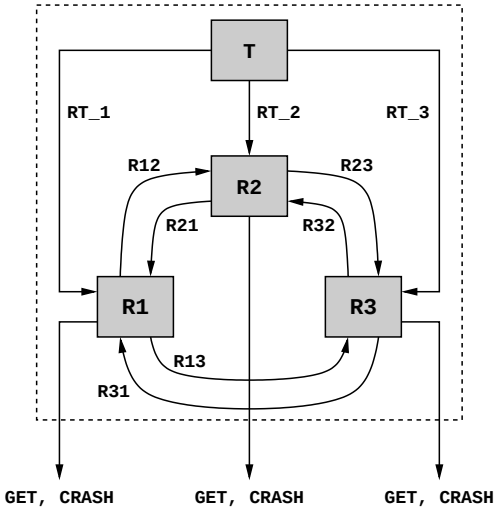

Fig. 3. Architecture of the rel/REL Protocol

3 Refined Compositional Verification

The basic compositional verification approach presented in Section 2 may fail because 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 [6, 7]. To overcome this problem, processes may be restricted w.r.t. so-called *interfaces* expressing the behavioral restrictions imposed on each process by synchronization with its neighbor processes.

Technically, an expression written "B `-|[`GL`]|` I", where I is the interface (an LTS), B the behaviour to restrict, and GL a set of gates named *synchronization set*, denotes the biggest sub-LTS of B of which states and transitions can be reached following observable execution sequences of I, where actions on gates in GL only are considered observable. "B `-||` I" is a shorthand notation for "B `-|[`GL`]|` I", where GL is the set of gates occurring in I. Interfaces can be given by the user (in which case their correctness must be checked) or generated automatically. The "`?`" symbol, possibly placed before the interface (see below), indicates that correctness of the interface w.r.t. the environment must be checked during state space construction.

To illustrate how SVL supports refined compositional verification, we use the reliable atomic multicast protocol [2] example presented in [7][3]. Figure 3 describes a protocol configuration consisting of one transmitter (process `T`) and three receivers (processes `R1`, `R2`, `R3`). This protocol can be represented by the following LOTOS and SVL like parallel composition expression:

```
hide R_T1, R_T2, R_T3, R12, R13, R21, R23, R31, R32  in
```

[3] SVL supersedes the DES2AUT tool described in [7]; see [4] for a comparison.

```
((R1 |[R12, R21, R13, R31]| (R2 |[R23, R32]| R3))
 |[R_T1, R_T2, R_T3]| T)
```

Direct generation of this behaviour would lead to state explosion. Instead, user-given interfaces "`r1.lotos`", "`r2.lotos`", and "`r3.lotos`" are used to restrict each receiver, and `T` is also used as an interface to restrict intermediate compositions. Verification is specified using the following SVL script :

```
% DEFAULT_LOTOS_FILE="rel_rel.lotos"
"T.bcg" = strong reduction of T;                                  (* 1 *)
"rel_rel.exp" = leaf strong reduction of                          (* 2 *)
   hide R_T1, R_T2, R_T3, R12, R13, R21, R23, R31, R32  in
   ((((R1 -||? "r1.lotos")
      |[R12, R21, R13, R31]|
      (((R2 -||? "r2.lotos") |[R23, R32]| (R3 -||? "r3.lotos"))
       -|[R_T2, R_T3]| "T.bcg"))
     -|[R_T1, R_T2, R_T3]| "T.bcg")
    |[R_T1, R_T2, R_T3]| "T.bcg");
"rel_rel.bcg" = strong reduction of "rel_rel.exp"                 (* 3 *)
```

In step (1), process `T` of "`rel_rel.lotos`" is generated and minimized for strong bisimulation. In step (2), for each `R`i an LTS is generated using the PROJECTOR tool of [7], by taking into account the restrictions specified by the corresponding "`r`i`.lotos`" interface. The resulting three LTSs are then minimized for strong bisimulation, composed in parallel (following the restrictions specified by `T`), and finally minimized for strong bisimulation. This produces an LTS which is composed in parallel with `T` to form (after hiding internal gates) a network of LTSs named "`rel_rel.exp`". Note that since `R_T1` does not occur in `R2` and `R3`, it obviously does not appear in the synchronization set of the restriction w.r.t. `T` of the `R2` and `R3` composition. In step (3), the LTS corresponding to "`rel_rel.exp`" is generated and minimized for strong bisimulation. During the state space construction, the correctness of interfaces preceded by "`?`" is checked automatically. The final LTS ($150,911$ states and $1,249,375$ transitions) is obtained in 15 minutes on a 450 MHz Linux PC.

4 Other Forms of Scripted Verification

Besides compositional verification, SVL is also convenient to perform other forms of verification (e.g., those based on bisimulations or temporal formulas) permitted by the CADP tools. For instance, the following script verifies that it is always possible to perform the "`S !1`" action from any state of the LTS "`f.bcg`". This is checked by hiding all labels but "`S !1`", then comparing modulo branching equivalence the resulting LTS to another LTS with a single state and a single looping transition labeled "`S !1`", contained in file "`r.bcg`".

```
"d.seq" = branching comparison
    (total hide all but "S !1" in "f.bcg") == "r.bcg";
```

By combining SVL with Bourne shell features, it is also possible to introduce parameterization in verification scenarios. The following script uses a "`for`" loop to verify eight temporal logic properties (contained in files "`prop1.mcl`", ..., "`prop8.mcl`") on the LTS "`f.bcg`". Lines starting with "`%`" are meant to be Bourne shell. Other shell control structures ("`if...fi`", "`case...esac`", function definitions, etc.) can be used similarly.

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

5 Conclusion

Scripting languages will certainly play a growing role in advanced verification tool sets. The SVL scripting language added recently to CADP makes compositional verification simpler than ever by interconnecting many verification tools and file formats transparently. Although very recent, SVL is already used in both academic and industrial projects, e.g., at the University of Twente (The Netherlands) and Ericsson (Sweden). Practical experiments (19 out of the 29 CADP demos have been rewritten in SVL) indicate that SVL leads to more readable, shorter, and safer scripts than equivalent MAKEFILEs and shell scripts.

References

1. A. Bouali, A. Ressouche, V. Roy, and R. de Simone. The Fc2Tools set: a Toolset for the Verification of Concurrent Systems. In *CAV'96*, *LNCS* vol. 1102.
2. S. Bainbridge and L. Mounier. Specification and Verification of a Reliable Multicast Protocol. Technical Report HPL-91-163, HP Labs, Bristol, 1991.
3. 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 *CAV'96*, *LNCS* vol. 1102.
4. H. Garavel and F. Lang. SVL: a Scripting Language for Compositional Verification. In *FORTE'01* (Kluwer) and INRIA Research Report RR-4223.
5. H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. INRIA Technical Report RT-0254, 2001.
6. S. Graf and B. Steffen. Compositional Minimization of Finite State Systems. In *CAV'90*, *LNCS* vol. 531.
7. J.-P. Krimm and L. Mounier. Compositional State Space Generation from LOTOS Programs. In *TACAS'97*, *LNCS* vol. 1217.
8. J. Romijn. Model Checking the HAVi Leader Election Protocol. Technical Report SEN-R9915, CWI, Amsterdam, 1999.