SVL: A Scripting Language for Compositional Verification

Hubert Garavel, Frédéric Lang

INRIA Rhône-Alpes / VASY

655, avenue de l'Europe

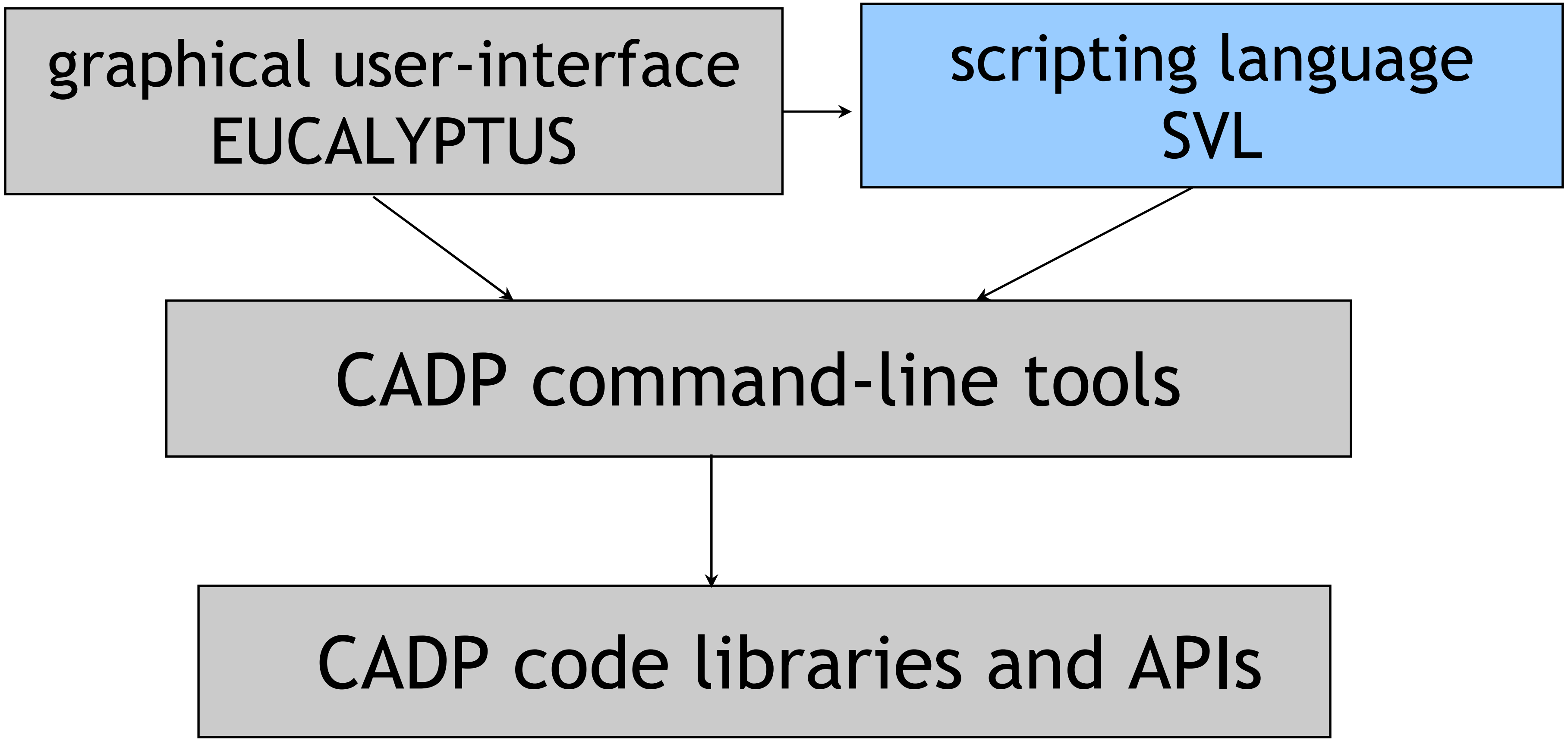
F-38330 Montbonnot Saint Martin

France

CADP (CAESAR/ALDEBARAN) Tools

- A toolbox for protocol engineering
- Various input languages:
  - LOTOS
  - networks of communicating automata
- Various intermediate models:
  - explicit LTSs (BCG)
  - implicit LTSs (Open/Caesar)
- Bisimulation (Aldébaran, Bcg_Min, Fc2Tools)
- Model-checkers (XTL, Evaluator 3.0)
- Simulation, rapid prototyping, test generation…

Interface: Graphics vs Scripts



Why Scripting ?

- Verification scenarios can be complex
- They can be repetitive
- Many objects/formats to handle:
  - High-level process descriptions (e.g., LOTOS)
  - Networks of communicating LTSs
  - Explicit and implicit LTSs
- Many operations to perform:
  - LTS generation of a LOTOS program, a network of LTSs
  - Label hiding, label renaming
  - LTS minimization/comparison modulo equivalences
  - Verification (deadlock, livelock, temporal logic formula)
- Various verification techniques:
  - enumerative, on-the-fly, compositional, etc.

What is SVL?

- An acronym: *Script Verification Language*
- A language for describing (compositional) verification scenarios
- A compiler (SVL 2.0) for executing scenarios writen in this language
- A software component of CADP 2001

Outline

- The SVL Language
- Compositional Verification in SVL
- The SVL Compiler

SVL Components

Two types of components can be mixed

- SVL verification statements (written S)
  - Compute and store an LTS or network of LTSs in a file
  - Verify temporal properties
  - Compare LTSs, etc.
- Bourne shell constructs (lines starting with %)
  - Variables, functions, conditionals, loops, …
  - All Unix commands

SVL Behaviours

- Algebraic expressions used in statements
- Several operators
  - Parallel composition
  - LTS generation and minimization
  - Label hiding and renaming, etc.
- Several types of behaviours
  - LTSs (four formats)
  - Networks of communicating LTSs (two formats)
  - LOTOS descriptions
  - Particular processes in LOTOS descriptions

Abstract Syntax of Behaviours

B ::= *"F.bcg"* | *"F.aut"* | *"F.fc2"* | *"F.seq"*

| *"F.lotos"* | *"F.lotos" : P [G_1, ..., G_n]*

| *"F.exp"*

| B_1 |[G_1, ..., G_n]| B_2 | B_1 ||| B_2

| *generation of* B_0

| *R reduction [using M] [with T] of* B_0

| *[S] hide [all but]* L_1, ..., L_n *in* B_0

| *[S] rename* $L_1 \rightarrow L_1'$, ..., $L_n \rightarrow L_n'$ *in* B_0

| *[user] abstraction* B_1 *[sync* G_1, ..., G_n*] of* B_2

Explicit LTSs

- States and transitions listed exhaustively
- LTSs in several formats

B ::=	*"F.bcg"*	Binary Coded Graphs
|	*"F.aut"*	Aldébaran ASCII format
|	*"F.fc2"*	Meije's FC2 format
|	*"F.seq"*	Set of traces

- Format conversions are fully automatic

CADP Tools for Explicit LTSs

- A set of tools to process BCG graphs
  - BCG_IO: Conversions from/to many other graph formats
  - BCG_MIN: Minimization for strong/branching bisimulation
  - BCG_LABELS: Label hiding and renaming
  - BCG_INFO: Display information about a graph
  - BCG_DRAW, BCG_EDIT: Draw/edit a BCG graph
- Aldebaran and the FC2 Tools
  - LTS minimizations/comparisons for several bisimulations

Implicit LTSs

- States and transitions given in comprehension
  - Initial state and transition relation
  - States generated *on-the-fly*
- Several types of implicit LTSs
  - LOTOS descriptions (*"F.lotos"*)
  - Particular LOTOS processes (*"F.lotos" : P* [G_1, ..., G_n])
  - Networks of communicating automata (*"F.exp"*)

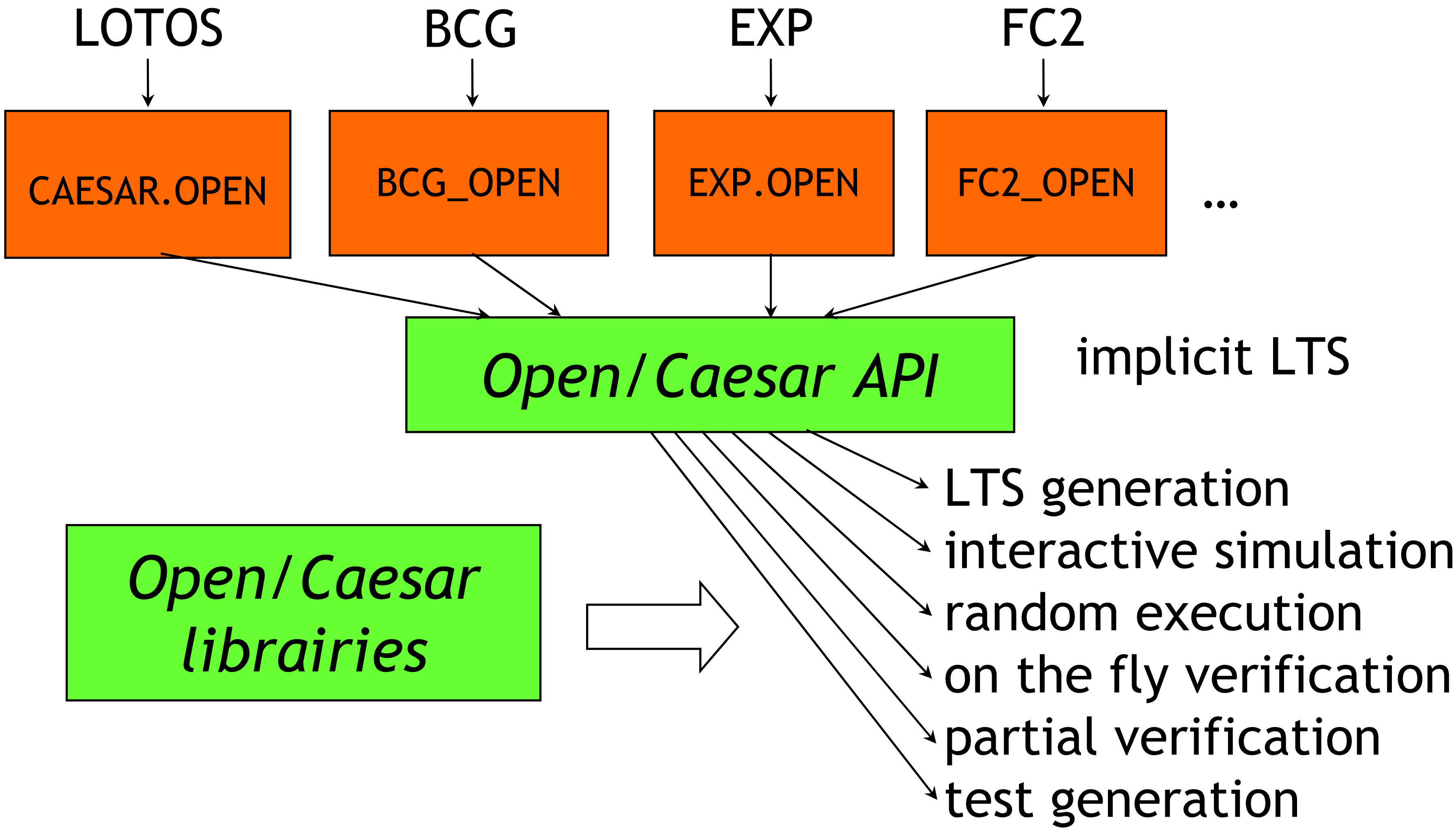
    LTSs combined with parallel and hiding, e.g.,

    hide G_1 *in ("spec*$_1$*.bcg"* |[G_1, G_2]| *"spec*$_2$*.aut")*
  - Parallel FC2 is also partly supported

CADP Tools for Implicit LTSs

- Special case: communicating LTSs
  - Aldebaran and Exp.Open handle EXP files using *on the fly* and BDD methods
  - Fc2 Tools handle parallel FC2 files using BDD methods
- General case: OPEN/CAESAR
  - Primitives to compute initial state and successors
  - Modular separation between language-dependent compilers and language-independent tools

OPEN/CAESAR


LOTOS
BCG
EXP
FC2
CAESAR.OPEN
BCG_OPEN
EXP.OPEN
FC2_OPEN
...
Open/Caesar API
implicit LTS
Open/Caesar librairies
LTS generation
interactive simulation
random execution
on the fly verification
partial verification
test generation

Explicit vs Implicit LTSs

SVL principles:

- Keep LTSs implicit as long as possible
  - Explicit LTS generation is expensive (state explosion)
  - Not all properties necessitate to explore the whole LTS
- Explicit LTS generation is done only if required explicitly by the user

LTS Generation

Conversion from an implicit LTS to an explicit LTS

$$B ::= \textit{generation of } B_0$$

Examples

- *generation of "spec.lotos"*
  Use CAESAR.ADT and CAESAR
- *generation of "spec.lotos" : P [G]*
  Use CAESAR.ADT and CAESAR (option -root)
- *generation of "spec.exp"*
  Use EXP.OPEN and Generator
- *generation of*
  (("$spec_1$*.bcg" |[*G_1*]| "*$spec_2$*.aut") ||| "*$spec_3$*.bcg")*
  Use EXP.OPEN and Generator

Parallel Composition

$$B \ ::= \ B_1 \ |[G_1, \ldots, G_n]| \ B_2$$

$$| \quad B_1 \ ||| \ B_2$$

$$| \quad B_1 \ || \ B_2$$

- Synchronization on $G_1, \ldots, G_n$ (LOTOS semantics)
- B_1 and B_2 can be LTSs, but also any SVL behaviour
- Generation of intermediate EXP files

Label Hiding

$$B ::= [S]\ hide\ L_1, \ldots, L_n\ in\ B_0$$
$$\mid\ [S]\ hide\ all\ but\ L_1, \ldots, L_n\ in\ B_0$$

- An extension of LOTOS hiding, where

  L is either

  - a gate name
  - a label string (e.g. "G !3.14 !TRUE")
  - a regular expression (e.g. "G !.* !TRUE")

  S ::= *gate* | *total* | *partial* is a *matching semantics* for regular expressions

  all but means complementation of the set of labels
- Tools used: BCG_LABELS or EXP.OPEN

Label Hiding: Examples

[gate] hide G, H in "test.bcg"

invokes BCG_LABELS (-hide) and returns an LTS in which labels whose gate is G or H are hidden

total hide "G ![AB]." in "test.bcg"*

invokes BCG_LABELS and returns an LTS in which labels matching "G ![AB].*" are hidden

partial hide G in "test.bcg"

invokes BCG_LABELS and returns an LTS in which labels containing G are hidden

Label Renaming

$$B ::= [S]\ rename\ L_1 \rightarrow L_1', \ldots, L_n \rightarrow L_n'\ in\ B_0$$

where

- each $L \rightarrow L'$ is a Unix-like substitution containing regular expressions
- S is a matching semantics

$$S ::= gate \mid total \mid single \mid multiple$$

- Tool used: BCG_LABELS

Label Renaming: Examples

[gate] rename G -> H, H -> G in "test.bcg"

invokes BCG_LABELS (-rename) and returns LTS in which gate G is renamed into H and H into G

total rename "G !A !TRUE" -> "A_TRUE" in "test.bcg"

invokes BCG_LABELS and returns an LTS in which label "G !A !TRUE" is renamed into A_TRUE

total rename "G !\(.\) !\(.*\)" -> "G \2 \1" in "test.bcg"*

invokes BCG_LABELS and returns an LTS in which offers of labels whose gate is G are swapped

Reduction (also Minimization)

LTS Minimization modulo an equivalence relation

B ::= *R reduction* [*using M*] [*with T*] *of* B_0

- Several relations *R*
  strong, *branching*, *observational*, *safety*, *tau*.a*, etc.
- Several tools *T*
  aldebaran, *bcg_min*, *fc2tools*
- Several methods *M*
  std, *bdd*, *fly*
- Tools used: Aldebaran, BCG_MIN, Fc2

Reduction: Examples

strong reduction of "test.bcg" [with bcg_min]

invokes BCG_MIN and returns an LTS
minimized for strong bisimulation

branching reduction of "test.bcg" with aldebaran

invokes Aldebaran and returns an LTS
minimized for branching bisimulation

observational reduction of "test.bcg" with Fc2tools using bdd

invokes Fc2Min using BDD and returns an LTS
minimized for observational equivalence

Abstraction

- LTS generation of B_2 abstracted w.r.t. interface B_1

  B ::= *abstraction* B_1 *of* B_2

  | *user abstraction* B_1 *of* B_2

- Equivalent syntax

  B ::= B_2 -|| B_1

  | B_2 -||? B_1

  where *?* has the same meaning as *user*

- Detailed in Section on Compositional Verification

Abstract Syntax of Statements

S ::= *"F.E" = B_0*

| *"F.E" = R comparison* [*using M*] [*with T*] *B_1 == B_2*

| *"F.E" = deadlock* [*with T*] *of B_0*

| *"F.E" = livelock* [*with T*] *of B_0*

| [*"F_1.E" =*] *verify "F_2.mcl" in B_0*

Assignment Statement

$S ::= "F.E" = B_0$

- *Computes* B_0 and stores it in file *"F.E"*
- Extension *E* tells the format for *"F.E"* (*aut*, *bcg*, *exp*, *fc2*, *or seq*, *but not lotos*)
- Principles:
  - Format conversions are implicit (BCG_IO)

    e.g. *"spec.bcg"* = *"spec.fc2"* is permitted
  - No implicit LTS generation

    If *E* is an explicit LTS format (i.e. all but *exp*)
    then B_0 must not denote an implicit LTS
    ⇒ generation must be used explicitly

Comparison of Behaviours

S ::= *"F.E"* = *R comparison* [*using M*] [*with T*] $B_1 == B_2$
| *"F.E"* = *R comparison* [*using M*] [*with T*] $B_1 <= B_2$
| *"F.E"* = *R comparison* [*using M*] [*with T*] $B_1 >= B_2$

- Compares B_1 and B_2 and stores the distinguishing path(s) (if any) in *"F.E"*
- Equivalence or preorders
- Several relations *R* and several methods *M*
- Several tools *T* (*aldebaran* or *fc2tools*)

Deadlock and Livelock Checking

S ::= *"F.E"* = *deadlock* [*with T*] *of* B_0

| *"F.E"* = *livelock* [*with T*] *of* B_0

- Detects deadlocks or livelocks using tool *T* (*aldebaran*, *exhibitor*, *evaluator*, or *fc2tools*)
- Results in a (set of) paths leading to deadlock or livelock states and stored in *"F.E"*
- Verification may be on-the-fly (Exhibitor or Evaluator with OPEN/CAESAR)

Temporal Property Verification

$$S ::= [\text{"}F_1.E\text{"} =]\ verify\ \text{"}F_2.mcl\text{"}\ in\ B_0$$

- Checks whether B_0 satisfies the temporal logic property contained in *"F_2.mcl"* (μ-calculus)
- May generate a diagnostic and store it in *"F_1.E"* (example or counter-example which explains the resulting truth value)
- Verification may be on-the-fly (OPEN/CAESAR and Evaluator)

Shell Constructs in SVL Scripts

Shell commands can be inserted (%)

- Direct call to Unix commands (“echo”...)
- Setting of SVL shell variables
  - *% DEFAULT_COMPARISON_METHOD=fly*
  - *% CAESAR_OPTIONS=-gc*
- Enables the use of all shell control structures
  - "if-then-else" conditional
  - "for" loop
  - function definitions
  - etc.

A Simple Example

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

"obs.seq" = observational comparison
      "bitalt.bcg" == (generation of "simple.lotos");

"dead.seq" = deadlock of "bitalt.bcg";

% for N in 1 2 3 4
% do
            verify "prop_$N.mcl" in "bitalt.bcg"
% done
```

Outline

- The SVL Language
- <u>Compositional Verification in SVL</u>
- The SVL Compiler

SVL Key Features for Compositional Verification

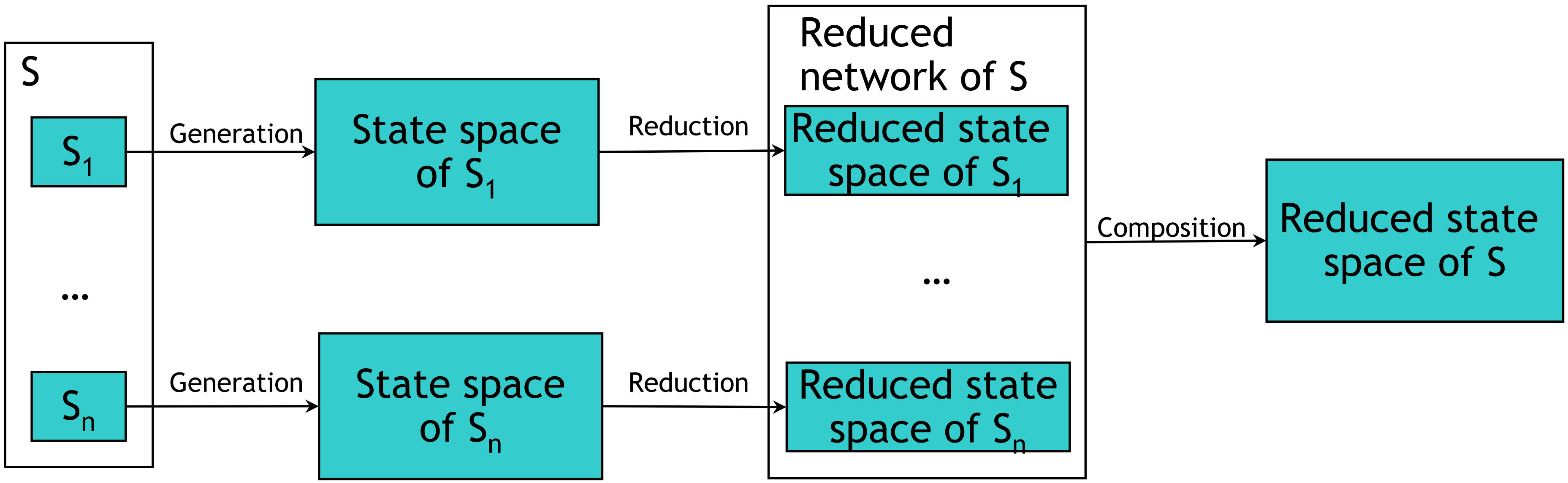
- Support for Basic Compositional Verification
  Example: The Alternating Bit Protocol
- Script Simplification using Meta-Operations
- Support for Refined Compositional verification
  Example: The rel/REL Protocol
- Compositional Performance Evaluation
  Example: The SCSI-2 Protocol

Basic Compositional Verification using CADP

How to avoid state explosion?

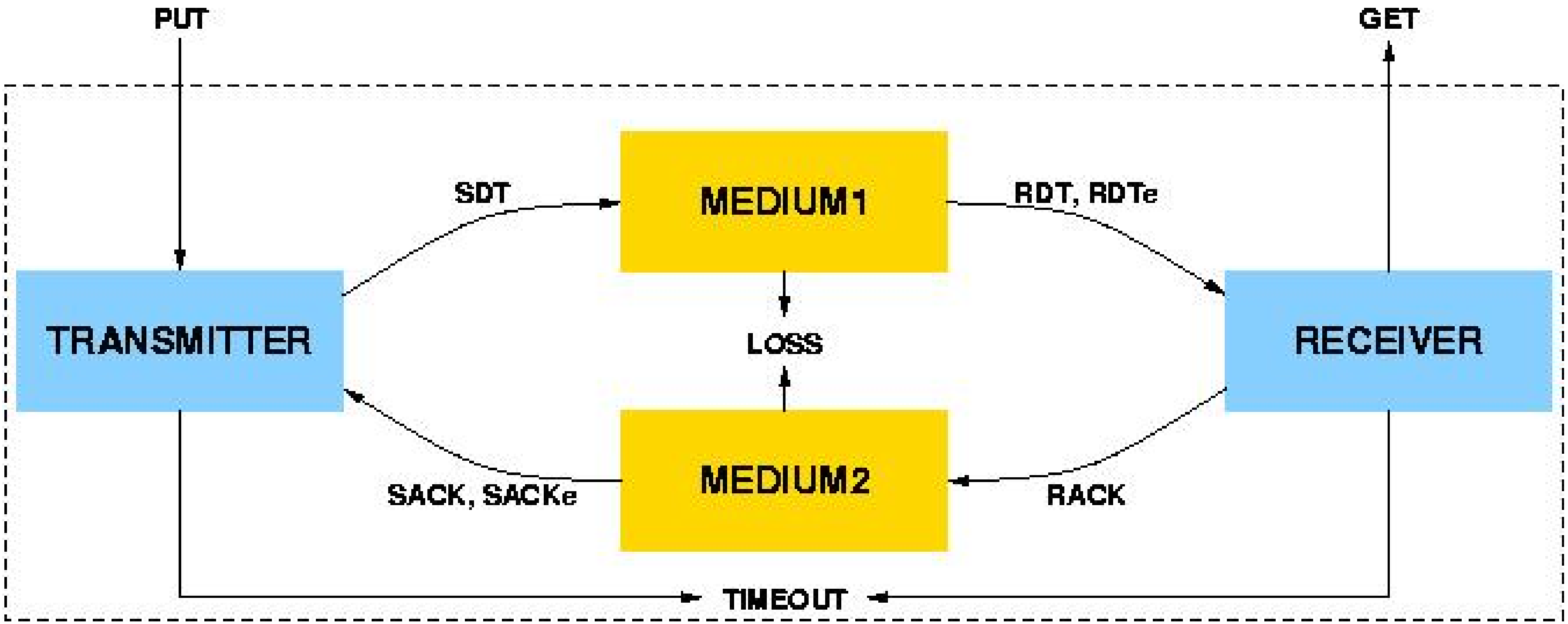
- Compositional generation: *"divide and conquer"*
  - Partition the system into subsystems
  - Minimize each subsystem modulo a strong or weak bisimulation preserving the properties to verify
  - Recombine the subsystems to get a system equivalent to the initial one
- CADP tools support this approach
  - Handle networks of minimized processes
  - On the fly or exhaustively

Basic Compositional Verification Illustrated



Example
The Alternating Bit Protocol (ABP)

Protocol architecture



Compositional Generation of the ABP using SVL

```
"bitalt.bcg" = strong reduction of
    hide SDT0, SDT1, RDT0, RDT1, RDTe, RACK0, RACK1, SACK0, SACK, SACKe in
     (
       (
          (strong reduction of "bitalt.lotos" : TRANSMITTER)
          |||
          (strong reduction of "bitalt.lotos" : RECEIVER)
       )
      |[SDT0, SDT1, RDT0, RDT1, RDTe, RACK0, RACK1, SACK0, SACK1, SACKe]|
       (
          (strong reduction of "bitalt.lotos" : MEDIUM1)
          |||
          (strong reduction of "bitalt.lotos" : MEDIUM2)
       )
     ) ;
```

Simplified ABP Script using the DEFAULT_LOTOS_FILE variable

```
% DEFAULT_LOTOS_FILE="bitalt.lotos"
"bitalt.bcg" = strong reduction of
   hide SDT0, SDT1, RDT0, RDT1, RDTe, RACK0, RACK1, SACK0, SACK1, SACKe in
      (
         (
            (strong reduction of TRANSMITTER)
            |||
            (strong reduction of RECEIVER)
         )
       |[SDT0, SDT1, RDT0, RDT1, RDTe, RACK0, RACK1, SACK0, SACK1, SACKe]|
         (
            (strong reduction of MEDIUM1)
            |||
            (strong reduction of MEDIUM2)
         )
      ) ;
```

Meta-operations

B ::= *leaf R reduction* [*using M*][*with T*] *of* B_0

| *root leaf R reduction* [*using M*][*with T*] *of* B_0

| *node R reduction* [*using M*][*with T*] *of* B_0

- Three compositional verification strategies:
  - Reduction of LTSs at the leaves of parallel compositions in B_0
  - Reduction of LTSs at the leaves of parallel composition in B_0 and then reduction of the whole behaviour
  - Reduction at every node of B_0
- Meta-operations expand to basic SVL behaviours

Simplified ABP Script using the "root leaf reduction" Meta-operation

```
% DEFAULT_LOTOS_FILE="bitalt.lotos"
"bitalt.bcg" = root leaf strong reduction of
   hide SDT0, SDT1, RDT0, RDT1, RDTe, RACK0, RACK1, SACK0, SACK1, SACKe in
      (
         (
            TRANSMITTER
            |||
            RECEIVER
         )
        |[SDT0, SDT1, RDT0, RDT1, RDTe, RACK0, RACK1, SACK0, SACK1, SACKe]|
         (
            MEDIUM1
            |||
            MEDIUM2
         )
      ) ;
```

Refined Compositional Verification

- Compositional verification may fail
  - Concurrent processes constrain each other
  - Separating tightly-coupled processes -> explosion
- Solution: use interfaces
  - [Graf-Steffen-91], [Krimm-Mounier-97]
  - Use interfaces to model the environment
  - CADP supports this approach
    - Projector tool (Krimm and Mounier)
    - Des2Aut tool: replaced by SVL

The Abstraction Behaviour

- The LTS of a behaviour *B* may be larger than the LTS of a behaviour containing *B* because of *context constraints*
- Example

  ("User1.bcg" ||| "User2.bcg") |[G]| "Medium.bcg"

  "Medium.bcg" may constrain the interleaving
- An SVL behaviour can be restricted w.r.t. an (exact or user-given) interface

$$B ::= [user]\ abstraction\ B_1\ of\ B_2$$

$$\mid \quad B_2\ \text{-}||[?]\ B_1$$

Use of Interfaces for Abstraction

- Interface = LTS understood as a set of traces
- Abstraction eliminates states and transitions of a process never reached while following all traces of its interface
- User-given interfaces involve predicate generation to check their correctness

SVL Example
The Rel/REL Protocol

```
% 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, R12, R21  in
            (
               (
                  abstraction (hide R_T1, R_T2 in "crash_trans.bcg") of
                     (
                        (user abstraction "r1_interface.lotos" of RECEIVER_NODE_1)
                        |[R12, R21]|
                        (user abstraction "r2_interface.lotos" of RECEIVER_NODE_2)
                     )
               )
               |[R_T1, R_T2, R_T3]|
               "crash_trans.bcg"
            );
```

Compositional Performance Evaluation

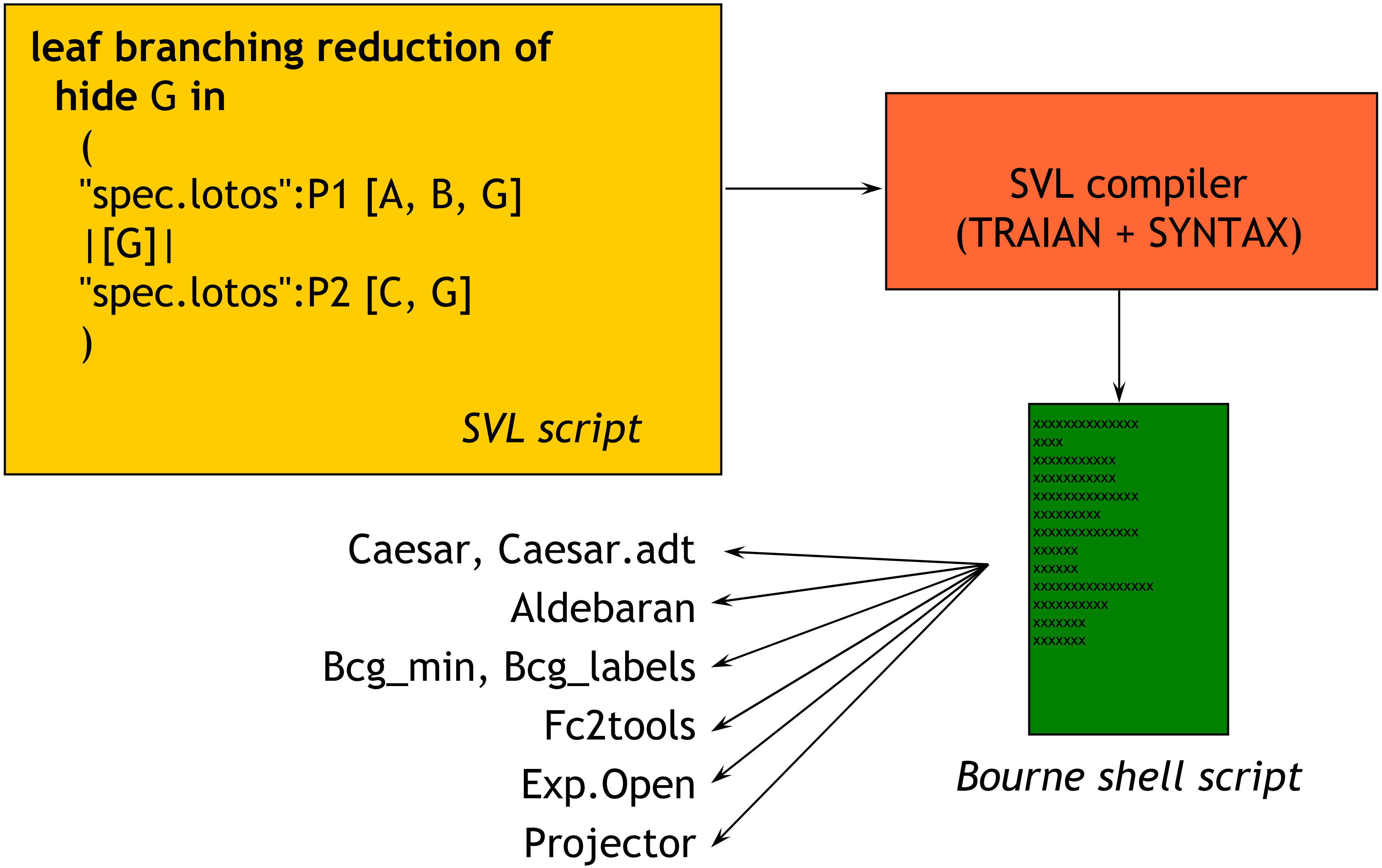
- SVL can also be used for compositional performance evaluation
- See FME 2002 paper by Garavel & Hermanns

  http://www.inrialpes.fr/vasy/Publications/Garavel-Hermanns-02.html

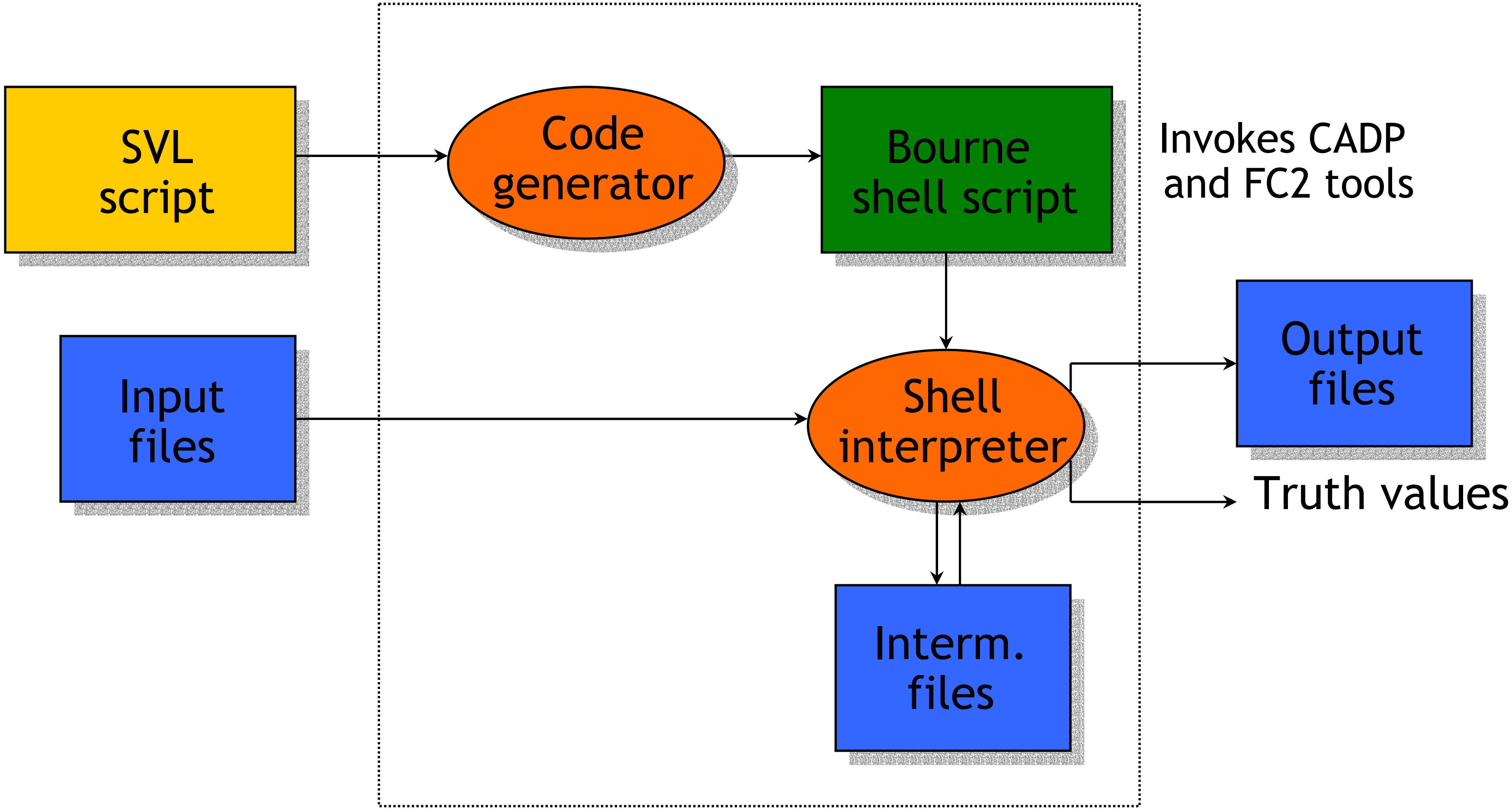
Outline

- The SVL Language
- Compositional Verification in SVL
- <u>The SVL Compiler</u>

The SVL 2.0 Compiler



The SVL 2.0 Compiler Detailed



The SVL 2.0 Compiler

- 7 000 lines of code
  (SYNTAX + LOTOS NT + C + Bourne Shell)
- Important design effort
  - Concise messages + log of execution
  - Erase intermediate files as soon as possible
  - Several modes to debug SVL descriptions
  - Implements «expert» knowledge
    (e.g., alternative reduction strategies)

Conclusion

- SVL is a new language and a new tool
- Fully integrated in CADP 2001
- Originally designed for compositional verification
- But now used for most CADP demos (27 over 31)
- Advantages
  - Avoids knowledge of each tool options/syntax
  - Avoids Makefiles, script-shells, intermediate files
  - Improves readability of verification scripts
  - A 5 page Makefile -> (much clearer) 2 page SVL script
- Extensible to support new tools
- Positive feedback from several users
- Compositional Verification becomes practical

More Information

- FORTE 2001 paper

  http://www.inrialpes.fr/vasy/Publications/Garavel-Lang-01.html

- TACAS 2002 tool paper

  http://www.inrialpes.fr/vasy/Publications/Lang-02.html

- SVL manual page

  http://www.inrialpes.fr/vasy/cadp/man/svl.html

- CADP and demo examples

  http://www.inrialpes.fr/vasy/cadp