Compositional Verification using CADP of the ScalAgent Deployment Protocol for Software Components

Frédéric Tronel, Frédéric Lang, Hubert Garavel

INRIA Rhône-Alpes / VASY
655, avenue de l'Europe
F-38330 Montbonnot Saint Martin

Outline

- Compositional verification with CADP
- The ScalAgent deployment protocol
- Automated modeling in LOTOS
- Verification of the protocol using CADP
- Conclusions

The CADP Toolbox

- *CAESAR/ALDEBARAN Development Package*
- A tool set for protocol engineering

  Developed since 1985
- Main features:
  - process algebra (LOTOS)
  - equivalence checking (bisimulations)
  - model-checking (modal μ-calculus)
  - code generation / rapid prototyping
  - simulation
  - test generation

LOTOS

- Formal Description Technique
- Standardized at ISO (1989)
- System defined in two parts
  - Data part: abstract data types (ACT One)
  - Control part: process algebra (CCS, CSP)
- Asynchronous parallelism
- Message passing synchronization on gates

Verification of LOTOS Models

- Check properties on an LTS (Labeled Transition System) generated from the LOTOS model
- Two representations of LTSs
  - Explicit representation (BCG):
    states and transitions enumerated and stored in a file
  - Implicit representation (OPEN/CAESAR):
    initial state + successor function computed on the fly
- Two ways to generate LTSs
  - Directly from the LOTOS model (CAESAR, CAESAR.ADT)
  - Using compositional methods

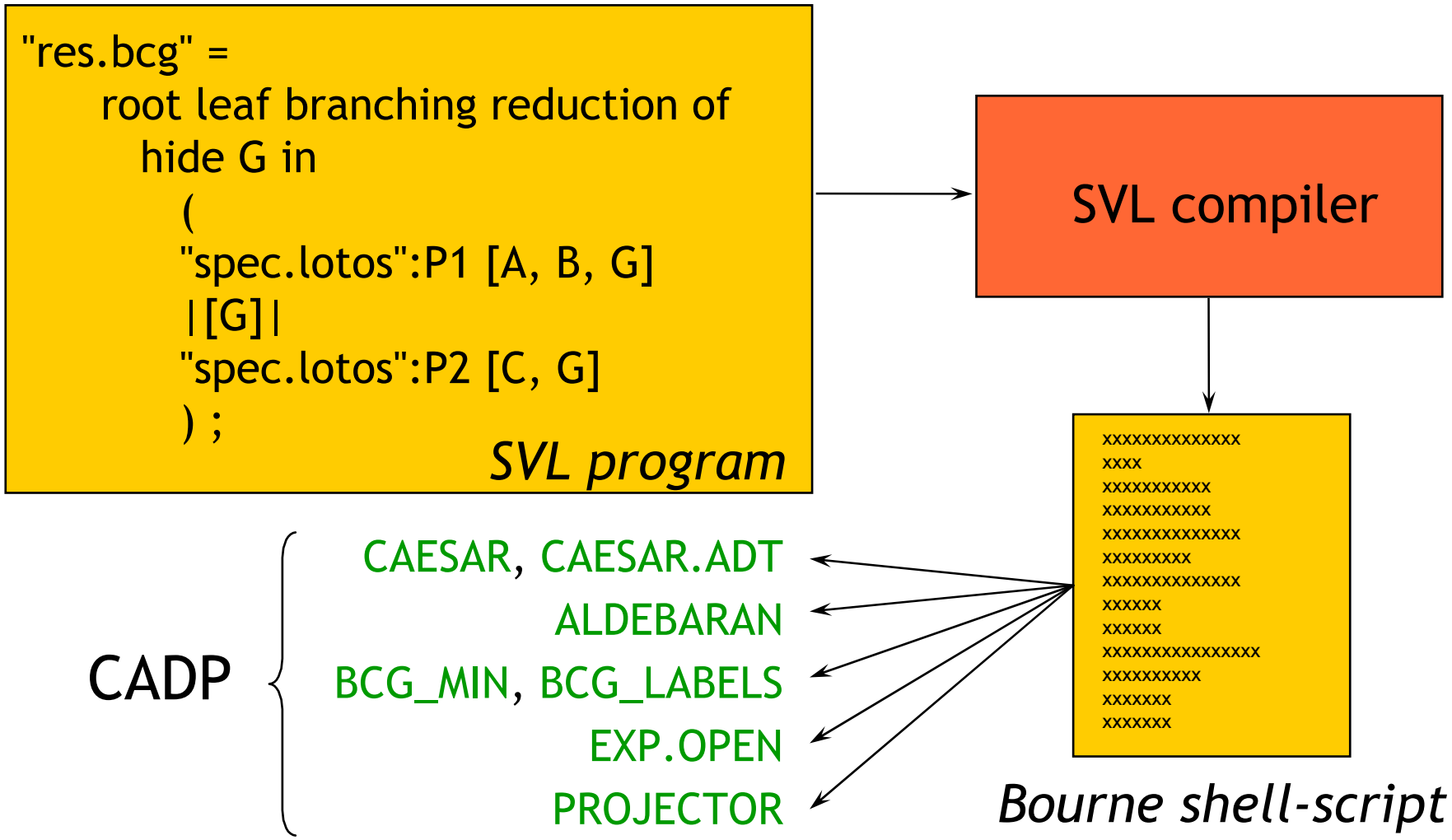
Compositional Verification in CADP

Assume system modeled as set of parallel LOTOS processes

- Generate the explicit LTS of each process separately (CAESAR, CAESAR.ADT), possibly using restriction interfaces (PROJECTOR)
- Apply abstractions preserving the properties to be checked (label hiding, minimization modulo bisimulation)
- Re-compose the parallel processes' LTSs to get the LTS of the whole system (EXP.OPEN)

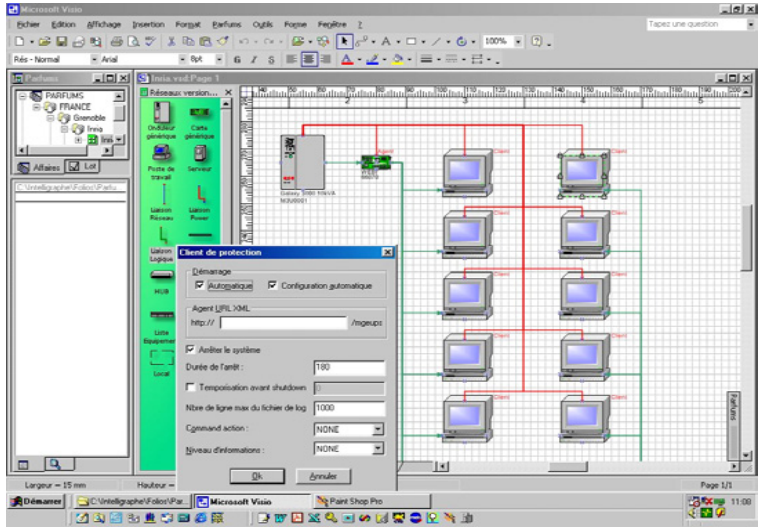
SVL allows to specify compositional verification scenarios in script files

SVL (Scripted Verification Language)



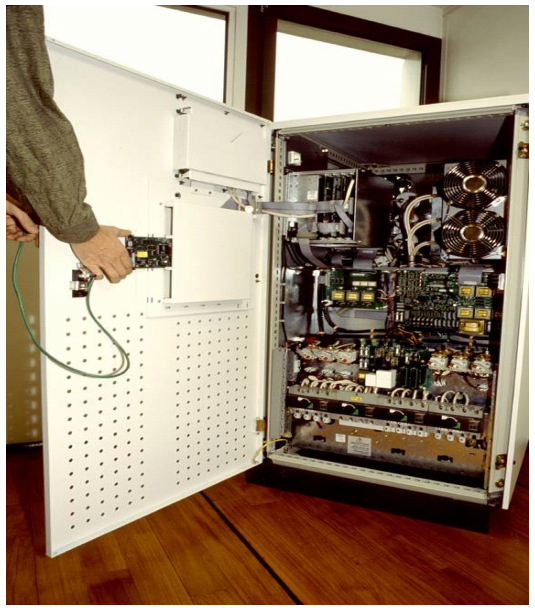
The ScalAgent Deployment Protocol

- Commercialized by the ScalAgent company
- Software deployment on heterogeneous devices
  - Deployed components written in any language
  - Deployment controlled by architecture specification (XML *configuration* file)



The ScalAgent Deployment Protocol

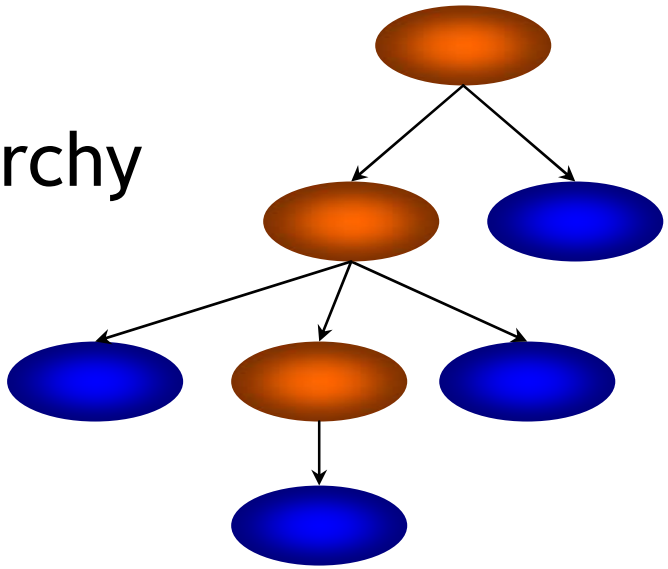
- Based on Java
  - Agents executing on geographically distributed JVMs
  - Local communication via method calls
  - Remote communication via the ScalAgent software bus
- Application to remote management of UPSs



A Tree Hierarchy of Java Agents

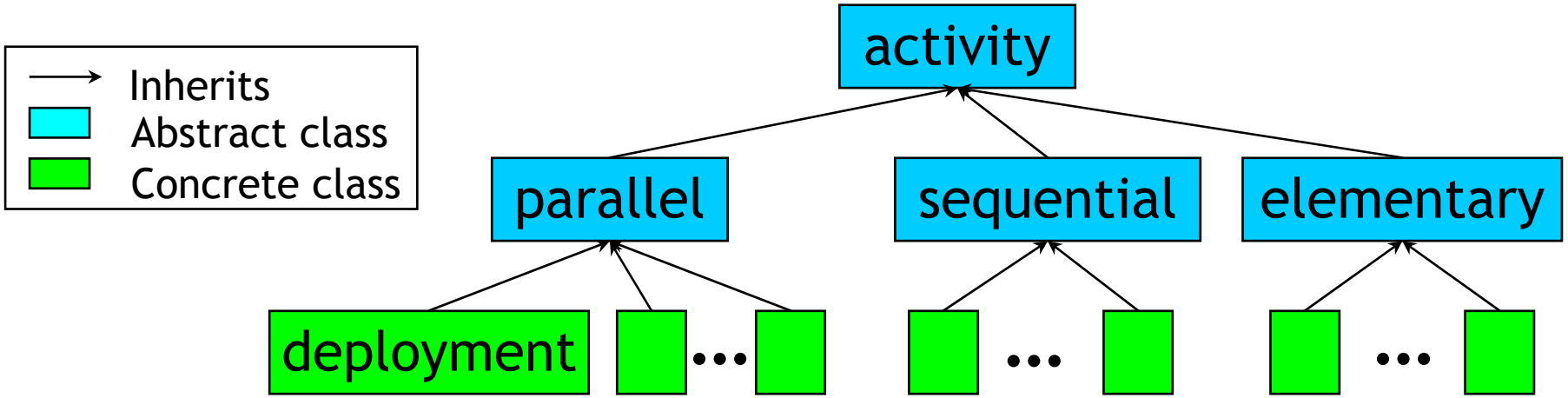
Hierarchy defined by 2 types of agents

- Container agents
  - Leaves of the tree hierarchy
  - Encapsulate the software components
- Controller agents
  - Higher nodes of the tree hierarchy
  - Manage the deployment



Definition of Agents

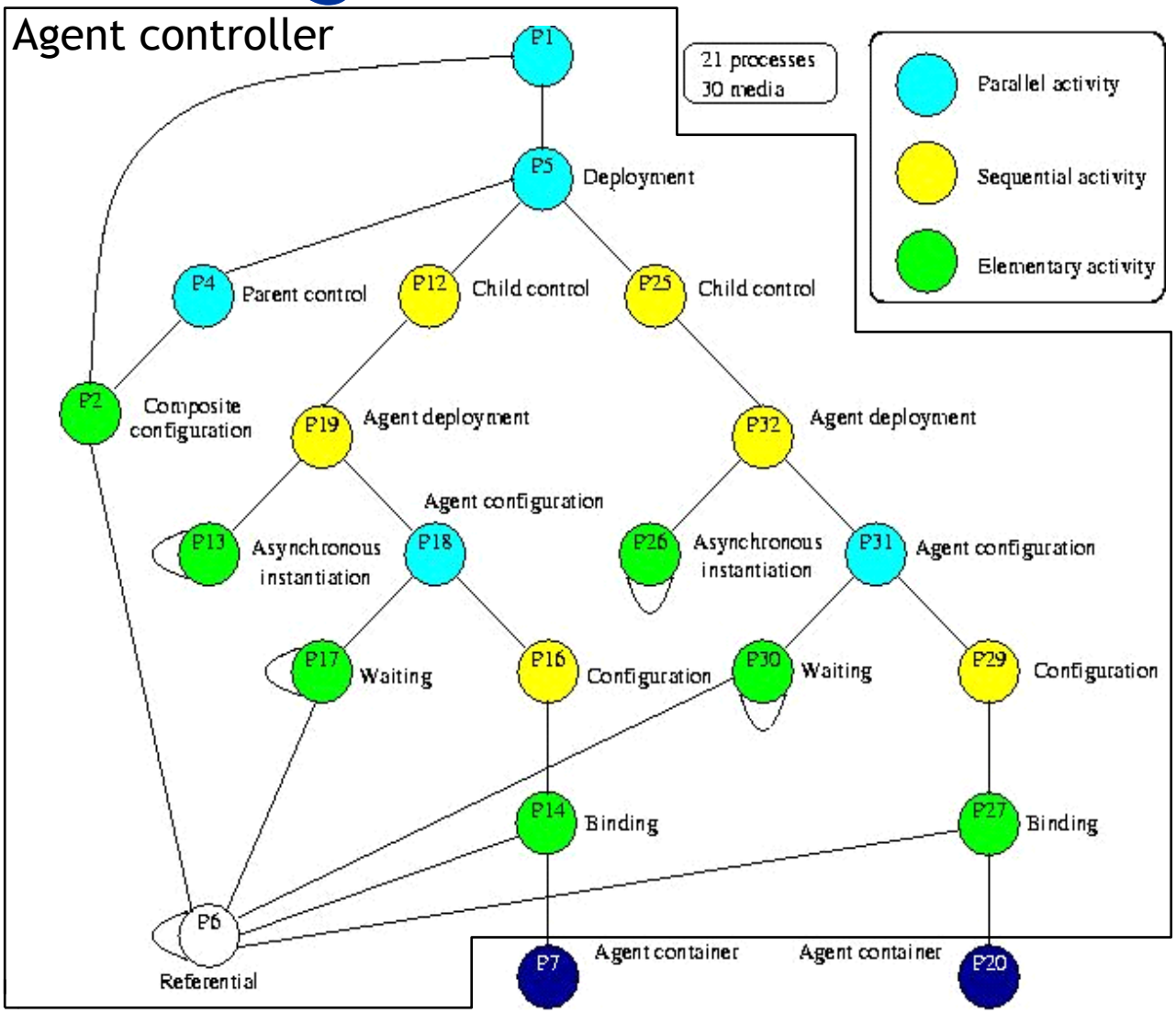
- Each agent runs a workflow (tree) of activities
- A Java class hierarchy of activities



- Sequential activities run sub-activities sequentially
- Parallel activities run sub-activities in parallel
- Only parent-child intra-agent communication

A Configuration

Unfolding of activities in the tree of agents



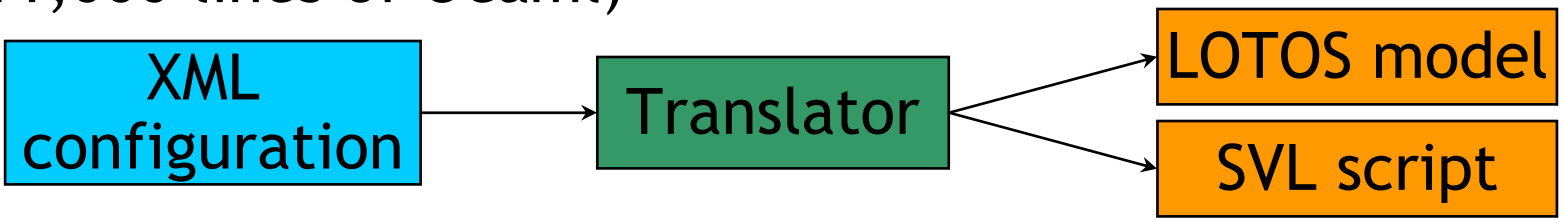
Activity Behaviour Definition

- Behaviour defined as event/reaction automaton
  - Event = message receipt
  - Reaction = message emissions and state change
  - Reaction depends on current state and message info: sender, type (start activity, end activity, …), etc.
- Behaviours are refined by inheritance

  Sub-class adds reactions to activity

Automated Modeling of Configurations

- Implementation of an automated translator
  (11,000 lines of Ocaml)



- The activity class hierarchy is reflected in the translator
  - For a given configuration, inheritance is resolved automatically by the translator
  - No inheritance resolution required in the LOTOS model

Generated LOTOS Model

- One LOTOS process for each activity
  - Gates SEND/RECV implement message exchanges
  - Also an ERROR gate to handle unspecified events
- 4 agents: 34 activities / 7,200 lines of LOTOS
- Model of communication (asynchronous) specified in the SVL script

Modeling Communications

- Communications can be modeled by finite process
  - Finite number of different messages
  - Bounded number of messages simultaneously in transit
- Centralized communication medium
  - One process managing all messages between activities
  - Not suitable to compositional verification (too large LTS)
- Distributed communication media
  - One process for each pair of communicating activities
  - Easy to generate in isolation (small LTS)

Compositional Verification using Distributed Media

- Incremental generation
  - Starting at the leaves: compose 2 LTSs of communicating activities with their medium
  - Hide communications local to these activities
  - Apply minimization wrt branching bisimulation
  - Repeat towards the root
- This allows to handle LTSs of tractable size
  - Always < 10^6 states in our experiments
  - Whereas potential global LTS of the order 10^{68} states

Communication Media Generation

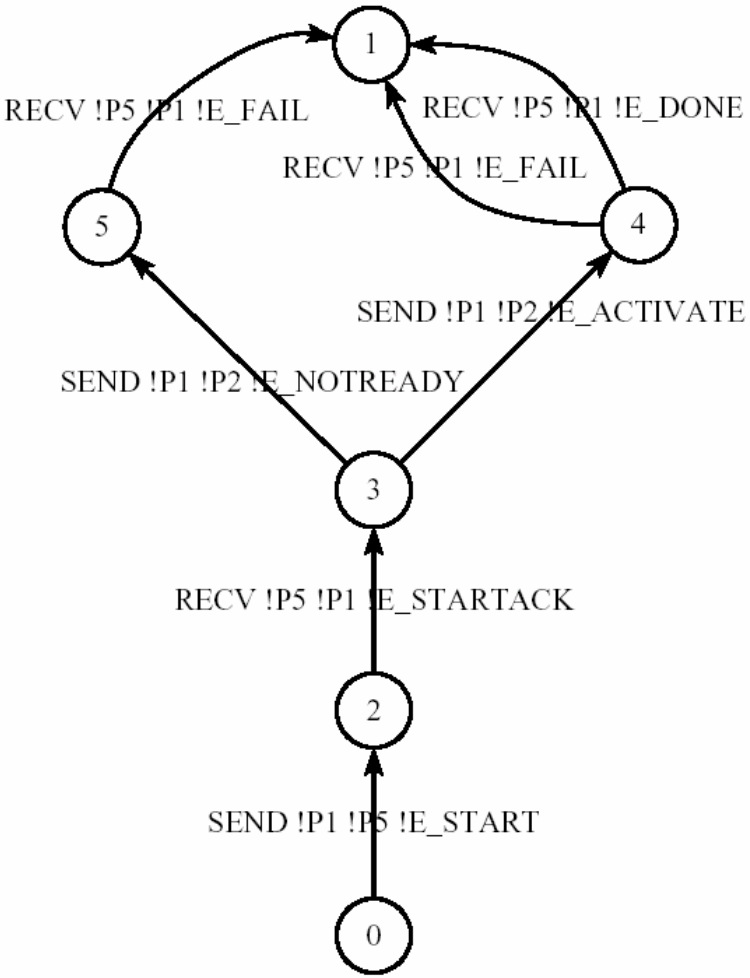
- Media sizes are not given in the specification
- Too small media may lead to deadlocks
- Generate media by successive tries:
  - Choose an arbitrary size for the medium and generate (using the BCG_GRAPH new tool)
  - Compose it with the corresponding activities
  - Check in composition whether overflow occurred; If yes, medium is enlarged and tried again

Excerpt of Generated SVL Script

```
% N=3
% while true; do

    "MEDIUM_19_18.bcg" = bag "$N" with "LABELS_19_18.txt"  ;

    "TMP.bcg" = branching reduction of
       gate hide all but RECV_19_18, SEND_19_18, RECV_18_19, SEND_18_19 in
         generation of
         ("CLUSTER_19_13.bcg" |[RECV_18_19, SEND_19_18]|
           ("MEDIUM_19_18.bcg" |[RECV_19_18, SEND_18_19]| "ACTIVITY_18.bcg")) ;

    "SUB_MEDIUM.bcg" = abstraction "TMP.bcg" of "MEDIUM_19_18.bcg " ;

%    RES=`bcg_open SUB_MEDIUM.bcg evaluator CHECK_$N.mcl | grep '\<TRUE\>'`
%    if [ "$RES" = "" ]; then
%       break
%    else
%       N=`expr $N + 1`
%    fi
% done
```

Verification Results

- Successful verification of large configurations
  - Up to 70 concurrent processes (activities + media)
  - Verified in less than 20 minutes
- Several impreciseness found in informal specification
- Absence of ERROR messages
- Size of distributed media is usually small (<10)
- Service provided by main controller is that expected



Service provided by the main controller (as generated from the LOTOS model)

Verification Data

Number of controllers	1	1	1	2
Number of containers	1	2	3	2
Total number of agents	2	3	4	4
Number of activities	13	21	29	34
Minimal size of activities (states)	7	7	7	7
Mean size of activities (states)	42	57	82	68
Maximal size of activities (states)	104	225	481	195
Number of media	18	30	42	36
Minimal size of media (states)	2	2	2	2
Mean size of media (states)	57	60	61	58
Maximal size of media (states)	111	111	111	111
Number of concurrent processes	**31**	**51**	**71**	**70**
Size of potential state space (states)	**2.10^{24}**	**3.10^{41}**	**4.10^{68}**	**9.10^{68}**
Size of largest generated LTS (states)	**1,824**	**48,819**	**410,025**	**76,399**
Size of generated LOTOS file (lines)	2,597	4,494	6,391	7,208
Size of generated SVL file (lines)	617	1,013	1,409	1,635
Number of intermediate files	221	316	503	519
Verification time	**4 min 09**	**9 min 52**	**19 min 43**	**12 min 10**

Conclusion

- Compositional verification is practical
  - Particularly adapted for systems with many components
  - Scale up better than non-compositional methods
- CADP compositional verification tools perform significantly better than direct methods
  - Abstraction by hiding and bisimulations
- SVL is a key ingredient of the success
  - No need to care about low-level details
  - Transparent management of hundreds of files (>500)

More on CADP

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