INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

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

Frédéric Tronel — Frédéric Lang — Hubert Garavel

N° 5012

Novembre 2003

THÈME 1

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--5012--FR+ENG

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

Frédéric Tronel* , Frédéric Lang† , Hubert Garavel‡

Thème 1 — Réseaux et systèmes
Projet VASY

Rapport de recherche n˚ 5012 — Novembre 2003 — 25 pages

Abstract: In this report, we present the application of the CADP verification toolbox to check the correctness of an industrial protocol for deploying and configuring transparently a large set of heterogeneous software components over a set of distributed computers/devices. To cope with the intrinsic complexity of this protocol, compositional verification techniques have been used, including incremental minimization and projections over automatically generated interfaces as advocated by Graf & Steffen and Krimm & Mounier. Starting from the XML description of a configuration of components to be deployed by the protocol, a translator produces a set of LOTOS descriptions, μ-calculus formulas, and the corresponding compositional verification scenario to be executed. The approach is fully automated, as formal methods and tool invocations are made invisible to the end-user, who only has to check the verification results for the configuration under study. Due to the use of compositional verification, the approach can scale to large configurations. So far, LOTOS descriptions of more than seventy concurrent processes have been verified successfully.

Key-words: automated verification, compositional verification, concurrency, distributed agents, objects, process algebra, protocol engineering, reachability analysis, software components

A short version of this report is available as "*Compositional Verification using CADP of the ScalAgent Deployment Protocol for Software Components*", in P. Stevens and U. Nestmann, editors, Proceedings of the 6th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems FMOODS'2003 (Paris, France), November 17-21, 2003.

This work was partially funded by the French Ministry of Industry under RNTL contract "PARFUMS".

* Frederic.Tronel@inria.fr
† Frederic.Lang@inria.fr
‡ Hubert.Garavel@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

Vérification compositionnelle avec CADP du protocole de déploiement de composants logiciels ScalAgent

Résumé : Dans ce rapport, nous présentons une application de la boîte à outils Cadp pour vérifier la correction d'un protocole industriel servant à déployer et configurer, de manière transparente, un grand nombre de composants logiciels hétérogènes sur un ensemble d'ordinateurs/appareils répartis. Pour faire face à la complexité intrinsèque de ce protocole, des techniques de vérification compositionnelle ont été utilisées, notamment des réductions incrémentales de l'espace des états et des projections sur des interfaces générées automatiquement selon les travaux de Graf & Steffen et Krimm & Mounier. Partant de la description en Xml d'une configuration de composants à déployer par le protocole, un traducteur produit un ensemble de descriptions Lotos, de formules de μ-calcul et le scénario de vérification à exécuter. L'approche est complètement automatisée, les méthodes formelles et les appels d'outils étant cachés à l'utilisateur final, qui doit seulement contrôler les résultats de la vérification pour la configuration étudiée. Grâce à l'utilisation de la vérification compositionnelle, l'approche peut traiter de grandes configurations. Pour le moment, des descriptions Lotos comprenant plus de soixante-dix processus concurrents ont été vérifiées avec succès.

Mots-clés : agents distribués, algèbre de processus, analyse d'accessibilité, composants logiciels, concurrence, ingénierie des protocoles, objets, vérification automatisée, vérification compositionnelle

1 Introduction

Formal verification methods are a key approach to establish the correctness of complex and critical object-oriented systems. This is true for sequential systems, and even more true for concurrent systems in which objects execute and interact using several threads of control.

However, the complexity of a system grows fast as the number of objects increases, so that attempts at verifying real-life systems are quickly confronted to the *state explosion* problem. It is therefore of crucial interest to focus on verification methods that scale up appropriately when applied to systems of increasing complexity.

Compositional verification methods usually follow a *divide and conquer* approach. The system to be verified is decomposed in several components, which are analyzed separately; afterwards, the results of the separate analyses are combined together to analyze the whole system. There are different approaches to compositional verification, depending whether the system under analysis is sequential or concurrent, and in the latter case, depending on the semantics used to model concurrency: linear-time or branching-time, state-based or action-based, etc. There is an important corpus of literature on compositional verification; for a survey, see for instance [19], [11, Section 1.1], [4, Sections 1.1 and 1.2], etc.

In spite of the many publications on compositional verification, the number of real-life case-studies in which compositional techniques have been applied is still low. This number is even lower if one considers the case of object-based systems, since compositional verification has been so far mostly used for communication protocols [20, 10, 14] or hardware protocols [3]. As for object-based systems specifically, one can mention several lines of work. [2] uses compositional proofs and refinement techniques to verify one-to-many negotiation processes for load balancing of electricity use. [1] uses a compositional proof system to verify correctness properties (expressed using the modal μ-calculus) for a set of applets executing on open platforms.

The present report is different, as it relies on enumerative (a.k.a., explicit state) model checking rather than proof techniques. It builds upon a prior application of compositional verification [6] to a dynamic reconfiguration protocol for a middleware agent-based platform. Using the CADP [9] verification toolbox, it was possible to establish the correctness of the reconfiguration protocol for several finite configurations determined by the number of agents, execution sites and protocol primitives. However, the approach did not scale well to larger configurations, mainly because the architecture of the system was specified in a centralized manner, all agents being connected to a central process modeling (an abstraction of) the software bus provided by the middleware infrastructure. This central process — more or less similar to a FIFO queue — prevented compositional verification from scaling up, as it was not possible to generate its entire state space in isolation from the remainder of the system. One key conclusion of [6] was the need for a more decentralized architecture specification in order to improve scalability.

Precisely, this research direction is addressed in the present report, still in the framework of industrial middleware infrastructures although on a different case-study than [6]. We consider here a deployment protocol for software components, which is commercialized by the

ScalAgent software company[1], and which we analyze using the compositional verification tools of Cadp.

The present report is organized as follows. Section 2 recalls the principles of compositional verification techniques and explains how they are supported within the Cadp toolbox. Section 3 describes the essential features of the ScalAgent deployment protocol. Section 4 gives hints of the formal modeling of the deployment protocol and indicates how the modeling task was, to a large extent, automated. Section 5 presents the main results of compositional verification. Finally, Section 6 concludes the report.

2 Compositional Verification with CADP

In this section, we first present enumerative and compositional verification techniques for systems composed of asynchronous processes, and we then detail how these techniques are supported by the Cadp verification toolbox.

Given a formal specification (e.g., using the Iso formal description technique Lotos [13]) of a concurrent system to be verified, enumerative verification relies on the systematic exploration of (some or all) possible executions of the system. The set of all possible executions can be represented as an Lts (*Labeled Transition System*), i.e., a graph (or state space) containing *states* and *transitions* labeled with communication actions performed by concurrent processes. There are two approaches to enumerative verification:

- In the first way, an *explicit* Lts is generated, i.e., states and transitions are first enumerated and stored, then analyzed by verification algorithms.
- In the second way, an *implicit* Lts (consisting of an initial state and a function that computes the successors of a given state) is constructed and verified at the same time, the construction being done *on the fly* depending on the verification needs. This allows to detect errors before the Lts has been generated entirely.

For complex systems, both approaches are often limited by the *state explosion* problem, which occurs when state spaces are too large for being enumerated. Two *abstraction* mechanisms are of great help when attacking state explosion:

- *Communication hiding* permits to ignore communication actions that need not be observed for verification purpose ;
- *Minimization* (with respect to various equivalence relations, such as strong bisimulation, branching bisimulation [21], etc.) allows to merge Lts states with identical futures and (possibly) to collapse sequences of hidden communication actions.

A further step is compositional verification, which consists in generating the Lts of each concurrent process separately, then minimizing and recombining the minimized Ltss taking

[1] `http://www.scalagent.com`

advantage of the congruence properties of parallel composition. The joint use of hiding and minimization to reduce intermediate state spaces enables to tackle large state spaces that could not be generated directly.

Although this simple form of compositional verification has been applied successfully to several complex systems (e.g., [3]), it may be counter-productive in other cases: generating the LTS of each process separately might lead to state explosion, whereas the generation of the whole system of concurrent processes can succeed if processes constrain each other when composed in parallel.

This issue has been addressed in various refined compositional verification approaches, which allow to generate the LTS of each separate process by taking into account *interface constraints* representing 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.

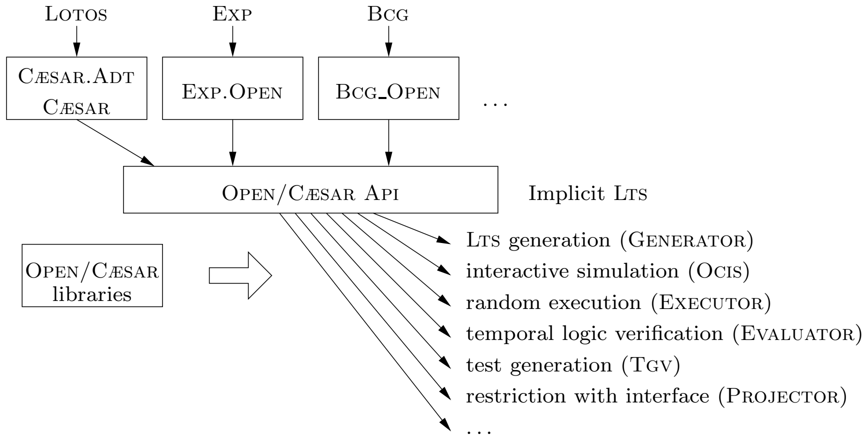


Figure 1: The OPEN/CÆSAR environment.

CADP[2] (*Construction and Analysis of Distributed Processes*) is a widely spread toolbox for protocol engineering, which offers a large range of functionalities, including interactive simulation, formal verification, and testing. CADP was originally dedicated to LOTOS, but its modular architecture makes it open to other languages and formalisms. CADP contains numerous tools for the compositional verification of complex specifications written in LOTOS:

- As regards explicit LTS representation, CADP provides a compact graph format, BCG (*Binary Coded Graph*) together with code libraries and tools to create, explore, and visualize BCG graphs, and to translate them from/to many other graph formats.

[2] `http://www.inrialpes.fr/vasy/cadp`

- As regards implicit LTS representation, CADP provides an extensible environment named OPEN/CÆSAR [7]. Although independent from any particular specification language, OPEN/CÆSAR has compilers for several languages: LOTOS (CÆSAR & CÆSAR.ADT), explicit LTSs (BCG_OPEN), networks of communicating LTSs (EXP.OPEN), etc. The corresponding architecture is illustrated in Figure 1. The code generated by OPEN/CÆSAR compilers is used by on the fly algorithms to perform simulation, verification, test generation, etc. on implicit LTSs.

- As regards LTS generation from LOTOS descriptions, CADP provides the CÆSAR and CÆSAR.ADT compilers, which can compile a LOTOS specification (or particular processes in this specification).

- As regards parallel composition of LTSs, CADP provides the EXP.OPEN compiler for handling networks of communicating LTSs, connected using LOTOS parallel composition and communication hiding operators.

- As regards communication hiding, CADP provides the BCG_LABELS tool, which allows to hide and/or rename the communication actions of an LTS.

- As regards LTS minimization, CADP contains two tools: BCG_MIN, which performs strong and branching minimization of LTSs efficiently, and ALDÉBARAN, which implements additional equivalences (safety equivalence, observational equivalence, and tau*.a equivalence) and LTS comparison algorithms.

- As regards generation with interface constraints, CADP provides the PROJECTOR tool, which implements the refined compositional verification approach of [12, 15]. PROJECTOR can be used to restrict LOTOS processes, explicit LTSs, as well as networks of communicating LTSs.

- As regards modeling of asynchronous communication media, we added a new tool named BCG_GRAPH, which generates various classes of useful LTSs, such as FIFO buffers and bags[3]. Distributed systems often contain many occurrences of such buffers that differ only by a few parameters, such as size and message names. Using BCG_GRAPH, very large buffers (several hundreds thousands states) can be generated quickly, with a small memory footprint.

- Finally, CADP includes a scripting language named SVL [8, 16], which provides a high-level interface to all the aforementioned CADP tools, thus enabling an easy description of complex compositional verification scenarios. For instance, the scenario consisting in generating separately the LTSs of three processes `P`, `Q`, and `R` contained in a file named `"spec.lotos"`, minimizing them for branching bisimulation, composing them in parallel, minimizing the resulting network on the fly, and storing the resulting LTS in the BCG format in a file named `"PQR.bcg"`, can be described by the simple SVL script that follows:

[3] A bag is a fully asynchronous buffer that does not preserve message ordering.

```
% DEFAULT_LOTOS_FILE="spec.lotos"
"PQR.bcg" = root leaf branching reduction of P || Q || R;
```

The SVL compiler translates such an SVL script into a Bourne shell script that, when executed, invokes CADP tools in the appropriate order and stores the results in intermediate files.

SVL has many additional features, namely operations to restrict a system with respect to a given interface, to minimize systems with respect to several other bisimulations and equivalences, to hide and rename communication actions, and statements to verify temporal logic formulas, to check for deadlocks and livelocks, and to compare systems with respect to a given equivalence or bisimulation. SVL scripts can also contain Bourne shell constructs, which enables to mix, e.g., conditionals, loops, and calls to Unix commands with SVL statements.

3 The ScalAgent Deployment Protocol

The work presented in this report was funded in the scope of PARFUMS (*Pervasive Agents for Reliable and Flexible Ups Management Services*), an industrial research project involving three companies (MGE-UPS, SILICOMP Research Institute, and SCALAGENT Distributed Technologies), and the VASY research group at INRIA.

The goal of PARFUMS is to solve problems of UPS (*Uninterruptible Power Supply*) management (installation, repair, and monitoring of remote equipments) in the case of large scale sites, by embedding software within UPSs (see Figure 2). To master the complexity induced by the distribution of applications, the project relies on the SCALAGENT platform for embedded systems, written in JAVA, to configure, deploy, and reconfigure software. Our contribution is about the modeling of the SCALAGENT deployment protocol and its verification using the CADP toolbox.

To ensure scalability, the SCALAGENT deployment protocol relies on a tree-like hierarchy of distributed agents communicating asynchronously by the mean of events. The tree of agents is meant to reflect the geographical distribution of software components to be deployed. The protocol uses two kinds of agents:

- *Containers* are located at the leaves of the tree hierarchy. They encapsulate software components written in any language, and act as interfaces with the rest of the protocol.
- *Controllers* are located at higher nodes of the tree hierarchy and manage the deployment. They only communicate with their parent and children agents, which allows a significant reduction of communications.

Controllers are specified by a workflow of *activities*, themselves structured as a tree, as shown in Figure 3. Activities fall into three categories, depending on the way they spawn subactivities:

Figure 2: Outside and inside view of an Ups (picture courtesy of Mge-Ups).

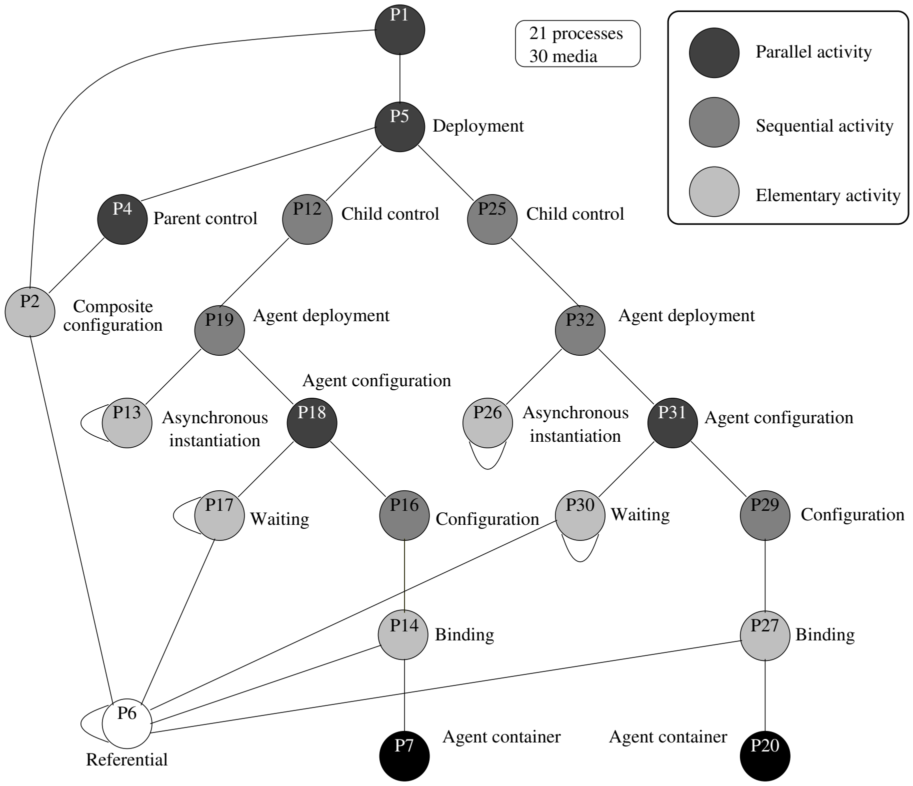


Figure 3: Process and communication structure for a configuration with one controller and two containers.

- *Elementary activities* are simple tasks that do not involve other subactivities. An example of such an activity is the receipt of a particular event from a container to signal its successful deployment, followed by an appropriate reaction.
- *Sequential activities* can spawn a set of subactivities, each one being executed after the previous one terminates.
- *Parallel activities* can spawn a set of simultaneous subactivities.

Beside activities, there exists a central *referential* process gathering information sent by the elementary activities regarding the success or failure of the deployment. Given this data, the referential process decides whether deployment is possible or not. The existence of communications between elementary activities and the referential "slightly breaks" the tree structure of communications. This is illustrated in Figure 3, the referential process being the white node in the "tree" of activities.

To describe distributed configurations, the ScalAgent infrastructure relies on the use of an Xml Dtd named Xolan. An Xolan configuration describes a set of controllers and containers, their geographical distribution, as well as their dependencies in terms of provided and required services, which rule the way the deployment must be performed. Xolan is generic, that is, not specific to Ups management. A graphical interface (see Figure 4) allows to specify a hierarchy of Upss and computers with the corresponding software to be deployed and generates the corresponding Xolan configuration automatically (see Figure 5).

4 Automated Formal Modeling of Configurations

It would have been possible to model Xolan configurations using Lotos abstract data types and to specify the deployment protocol as a Lotos process parameterized by the configuration to be deployed. However, this would have required the dynamic creation of processes in function of the configuration, which is not supported by mainstream enumerative verification tools.

Instead, we chose an automated approach by developing a translator (11,000 lines of the object-oriented, functional language Ocaml [17]) that takes an Xolan configuration, and produces both the Lotos specification corresponding to this configuration and an Svl script to perform the verification. This approach meets several requirements:

- Dynamic creation of processes is avoided, since the Ocaml translator can statically determine the set of processes created by the protocol for a given configuration.
- Xml parsing and Xolan data structure handling are delegated to the Ocaml translator rather than being coded as Lotos abstract data types.
- Even for simple configurations, the corresponding Lotos specifications and Svl scripts are complex, due to the large number of concurrent processes. The existence of an automated translator allows to propagate changes in the protocol modeling to each configuration under study so as to maintain consistency.

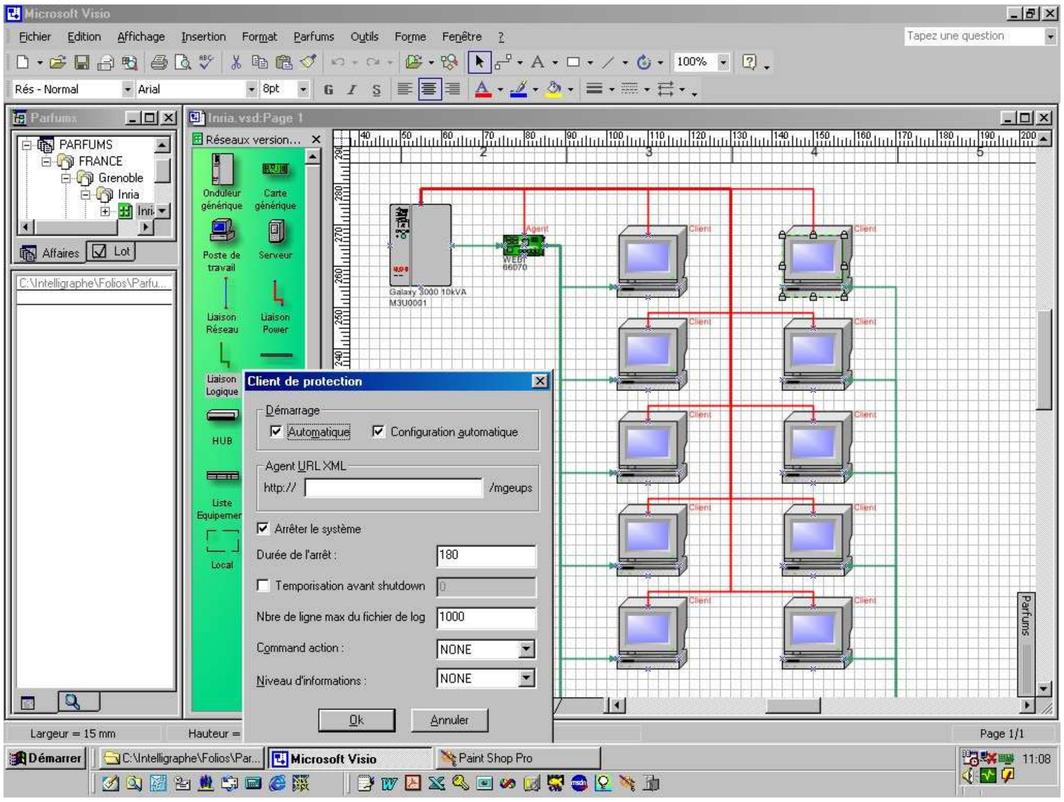


Figure 4: Graphical front-end for describing an UPS/computer hierarchy (picture courtesy of SILICOMP).

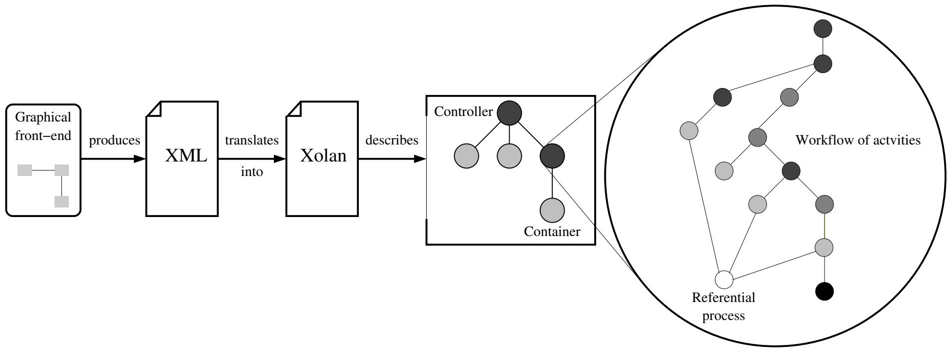


Figure 5: Different description levels for the deployment protocol.

To keep the formal verification of the protocol as intuitive as possible, each activity in the specification is translated into a separate process in the generated LOTOS code. This way, an incorrect behaviour in a given process can be immediately tracked back to the corresponding activity.

The processes generated for all activities share a similar form shown in Figure 6. Each process communicates with other parts of the system using three gates named `SEND`, `RECV`, and `ERROR`:

- "`SEND !from !to !event`" is used by the process whose identifier is stored in variable "`from`". It indicates that message "`event`" should be sent to the process whose identifier is stored in variable "`to`".
- "`RECV ?from:PID !to ?event:EVENT`" is used by the process whose identifier is stored in variable "`to`". Once the receipt is done, the variable "`from`" of type `PID` will contain the identifier of the sending process and the variable "`event`" of type `EVENT` will contain the received event.
- "`ERROR !from !number`" is used by the process whose identifier is stored in variable "`from`" to indicate that the error referenced as "`number`" has occurred. Each "`ERROR`" occurring in the LOTOS code has a different "`number`" that can be used to track down the errors.

Each process is recursive and stores its current state in a parameter "`state`" of type `STATE`. Each computation step of a process consists of message receipt, followed by some reaction depending on the identity of the message sender, the event received, and the current process state (the LOTOS construct "`[...]  -> ...`" reads as "*if ... then ...*"). A reaction consists in sending zero or more messages, followed by a state change, which is expressed by a recursive process call with actualized state. Some combinations of sender, event, and state may trigger an error message.

```
process P1 [SEND, RECV, ERROR] (state:STATE) : noexit :=
   RECV ?from:PID !P1 ?event:EVENT ;
   (
      [from eq P3] -> (
         [event eq E_START] -> (
            [state eq INIT] -> (
               SEND !P1 !P4 !START ;
               SEND !P1 !P6 !START ;
               BehaviourP1 [SEND, RECV, ERROR] (RUN)
            )
            []
            [not (state eq INIT)] -> ERROR !P1 !E1 ; stop
         )
         []
         [event eq E_STOP] -> ( ... )
         []
         [not ((event eq E_START) or (event eq E_STOP))] -> ERROR !P1 !E2 ; stop
      )
      []
      [from eq P5] -> ( ... )
      []
      [not ((from eq P3) or (from eq P5))] -> ERROR !P1 !E3 ; stop
   )
endproc
```

Figure 6: A LOTOS process following the asynchronous communicating process model.

The ScalAgent protocol specification is strongly object-oriented as it was written to prepare the way for a Java implementation of the protocol. All activities belong to an abstract "activity" class, which is refined into three abstract subclasses corresponding to elementary, sequential, and parallel activities respectively. Each of these abstract subclasses has itself concrete subclasses (for instance, deployment activities are a subclass of parallel activities).

The behaviour of an activity is a transition function obtained by combining the attributes specific to this activity (such as the number of subactivities for sequential and parallel activities, a unique identifier of the activity, a list of possible events, etc.) with the methods belonging to the activity class or transitively inherited from superclasses. Inheritance has the effect of adding reactions to new combinations of process parameters (events, states, process identifiers, etc.). For a given configuration of the deployment protocol, inheritance can be solved at compile-time. Thus, the Lotos process corresponding to an activity can be synthesized statically from the methods defined in the activity class and superclasses. The use of an object-oriented language such as Ocaml avoids the need for writing an inheritance resolution algorithm explicitly: by implementing the activity class hierarchy with a similar Ocaml class hierarchy, the resolution of inheritance is automatically performed by the Ocaml compiler.

5 Compositional Verification of the Protocol

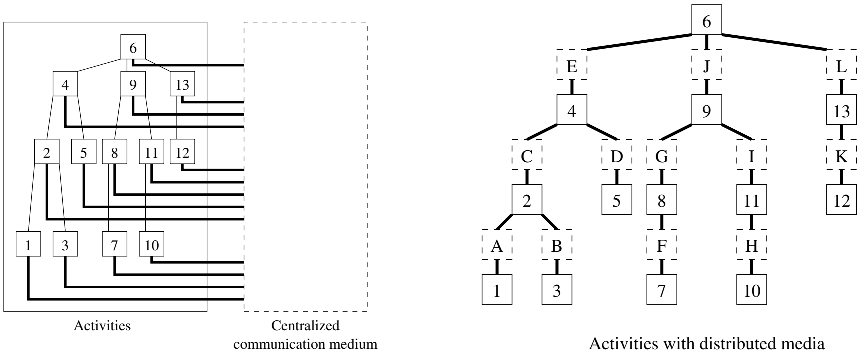
Compositional verification consists in decomposing the system under verification into smaller components that can be analyzed in isolation. There are often several ways of modeling and decomposing a given protocol; the feasibility and efficiency of compositional verification crucially relies on a thorough study of components and their interactions. In this section, we explain our choices and their impact on verification.

5.1 Centralized vs. Distributed Communication Media

By design, the ScalAgent deployment protocol ensures that communications are pairwise between an activity and each of its subactivities, and that messages do not accumulate indefinitely in communication media (i.e., Fifo buffers and bags). Therefore communication media can be described as finite processes containing a bounded number of messages belonging to a finite set of possible values.

The protocol specification leaves a degree of freedom in the implementation of communication media: it does not specify whether communications are conveyed using one unique centralized communication medium or several distributed communication media.

A system with a centralized medium is schematically depicted in Figure 7(a). This was the approach followed in [6] to model a software bus between agents. Unfortunately, this approach cannot be reused for the deployment protocol. As the number of activities increases, the number of different messages that can be exchanged increases as well. As more activities introduce more asynchrony in the system, the number of messages that must



(a) Activities communicating through a centralized medium.

(b) Activities communicating through distributed media.

Figure 7: Centralized vs. distributed media. Thick lines represent communications between activities and media, and thin lines represent the tree structure of activities.

be kept inside the medium also increases. Consequently, the state space of the centralized medium holding these messages may become too large for being generated separately.

In a refined approach, we split the centralized medium into one medium for each pair of communicating activities, as schematized in Figure 7(b). Each medium has to manage only a limited amount of communications, and is therefore less complex than the centralized medium.

This approach fits well with compositional verification, because the synchronizations between communicating processes are taken into account earlier in the verification process, leading to more constrained state spaces. Additionally, this permits to hide communications earlier, which, combined with minimization, gives greater opportunities to obtain reductions.

As an example, the architecture of Figure 7(b), can be generated incrementally by generating the state spaces of medium A, activity 1, and activity 2 first, then composing them together with appropriate synchronizations. Then, the local communications between 1 and 2 (exchanged via medium A) can be hidden, and the resulting LTS minimized for branching bisimulation. The resulting system is then composed with activity 3 and medium B, hiding appropriate actions and minimizing the resulting LTS. This way, by incorporating media and processes in the numbering order, the system can be generated up to the root of the activity tree. The progressive application of hiding and minimization steps allows to keep a state space of tractable size in spite of the complexity introduced by parallel composition.

5.2 Communication Media Generation

In general, bounding the size of a communication medium may cause unexpected deadlocks or lost behaviours because of buffer overflows.

We addressed the issue by first generating (using Bcg_Graph) a medium of limited size (say, $N = 3$ places), which is composed in parallel with its connected activities. This parallel composition is then used as an interface to restrict (using Projector) the behaviour of the medium itself. This produces a subset of the state space corresponding to the medium, in which only the transitions synchronized with the activities are kept.

We then check whether the N places of the medium have been used in the composition with its related activities. This is done by checking (using the Evaluator model checker) whether there exists a sequence of N successive messages received by the medium. If not, no buffer overflow has occurred, which means that the buffer size was bounded correctly. Otherwise, an overflow might have occurred, and the experiment must be restarted after incrementing the value of N.

This method should normally converge unless one medium appears to be unbounded, which would indicate either a protocol design error or a modeling error. The compositional approach permits to localize precisely such errors, which would be more difficult using a centralized medium.

Figure 8 shows a fragment of Svl script implementing this technique. Svl statements are intertwined with Bourne shell statements (starting with the % character). Figure 9 shows the information displayed to the user while the Svl script executes.

Although communication media could be expressed in Lotos and translated to Ltss (as for the activities), the use of the dedicated Bcg_Graph tool shortens the medium generation time by a factor of 10 to 100. This has a great impact on the overall verification time, since both the distributed media approach and the above technique to determine buffer size incrementally, require a significant number of media to be generated.

5.3 Additional Compositional Verification Tactics

More tactics have been used to make verification tractable:

- The referential process mentioned in Section 3 has been used as an interface to restrict (using the Projector tool) the behaviour of elementary activities communicating with this process. This divides by 2 the state space of some elementary activities. The referential process has been also used to restrict compositions of activities in several places.

- The state space of a process isolated from its context may explode if data communications are broken off without caution. For instance, the state space of the parallel composition of actions "$G\ ?X : \mathbf{nat} \mid\mid G\ !1$" has a single transition labeled "$G\ !1$". However, the state space of "$G\ ?X : \mathbf{nat}$" taken isolately cannot be generated because infinitely many different natural numbers can be received on gate G. For the deployment protocol, it is possible (though not easy) to determine statically the values

```
% N=3
(* we know empirically that many media have at least 2 places
 * we start from N=3 (instead of N=1) to save time *)

% while true
% do

(* generation of a bag with $N places between processes 18 and 19 *)
"MEDIUM_19_18.bcg" = bag "$N" using "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";

% echo -n "checking if a bag medium with $N places is large enough: "

(* using the Evaluator model-checker of CADP, we check if SUB_MEDIUM.bcg
 * contains a sequence of $N consecutive "SEND_xx_yy" actions *)

% RES=`bcg_open SUB_MEDIUM.bcg evaluator CHECK_$N.mcl | grep '\<TRUE\>'`
% if [ "$RES" = "" ]
% then
%    echo "yes"
%    break
% else
%    echo "no! (retrying with a larger bag medium)"
%    N=`expr $N + 1`
% fi

% done
```

Figure 8: An excerpt of the generated SVL script.

```
"MEDIUM_19_18.bcg" = bag 3 using "LABELS_19_18.txt"
      (* 286 states, 1320 transitions, 7.2 Kbytes *)

"svl094.exp" =
      "CLUSTER_19_13.bcg"
      |[RECV_18_19, SEND_19_18]|
      (
         "MEDIUM_19_18.bcg"
         |[RECV_19_18, SEND_18_19]|
         "ACTIVITY_18.bcg"
      )

"svl095.bcg" = generation of "svl094.exp"
      (* 985 states, 3508 transitions, 10.8 Kbytes *)

"svl096.bcg" = hide all but RECV_19_18, SEND_19_18, RECV_18_19, SEND_18_19 in
      "svl095.bcg"

"TMP.bcg" = branching reduction of "svl096.bcg"
      (* 129 states, 361 transitions, 3.9 Kbytes *)

"svl099.aut" = safety reduction (* with hiding *) of "TMP.bcg"
      (* 17 states, 24 transitions, 0.8 Kbytes *)

"SUB_MEDIUM.bcg" = abstraction "svl099.aut"
      of "MEDIUM_19_18.bcg"
      (* 13 states, 24 transitions, 3.0 Kbytes *)

checking if a bag medium with 3 places is large enough: yes

...
```

Figure 9: A run of the SVL script of Figure 8.

exchanged by a process on gates `SEND` and `RECV`. We thus have improved the generated LOTOS code by adding communication guards (synthesized during a first phase of the translation) to constrain the set of potentially received data.

5.4 Results of Compositional Verification

The study of the protocol allowed to clarify (in accordance with the SCALAGENT designers) several obscure points in the protocol specification. In particular, the original specification was silent about the model of communications between activities. Model checking verification revealed (by exhibiting an infinite loop of error messages) that the protocol was not meant for handling asynchronous messages between the inner activities of a controller, and would function properly only if communications inside a controller are implemented as local procedure calls (i.e., the calling activity gets suspended until the procedure call returns). Consequently, communications within a controller can be modeled as FIFO buffers instead of bags. However, bags are still needed to model communications between controllers, which can be geographically distributed.

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

Figure 10: Collected data for several configurations of the deployment protocol.

Figure 10 summarizes the verification results for four configurations. All experiments were done on a LINUX workstation with 1Gb memory and 2.2 GHz PENTIUM IV processor. We draw two main conclusions from these experiments:

- Although the high number of concurrent processes could lead to a potentially huge state space (up to 9.10^{68} states if we estimate its size by multiplying the numbers of states of the minimized Ltss corresponding to all activities and media for a given configuration), compositional verification allows to keep the state space to a tractable size (below 10^6 states).
- Given the large number of intermediate files (several hundreds), these experiments would not have been possible without the Svl language and associated compiler.

The correctness of each protocol configuration was determined by checking in the global Lts the absence of `ERROR` messages (which could denote either protocol design errors or implementation errors in the Ocaml translator). When the Lotos process corresponding to an activity is generated without its environment, all possible `ERROR` messages can be observed. However, when the process gets synchronized with all its children processes, there should not remain any `ERROR` message tagged with this process identity. For each configuration, we obtained the Lts of Figure 11, in which all communications are hidden but those made by the root activity of a controller. This Lts summarizes the service provided by the protocol to the end-user. The initial state has number 0. The two first transitions start the deployment by sending a start event and its acknowledgement. Then, either the user indicates that the deployment is not ready for activation, which will cause a failure notification from subactivities prevented from deploying components, or the user activates the deployment and receives either a notification that the deployment is successful or a failure indication.

6 Conclusion

Considering the numerous publications dealing with compositional verification and its expected benefits, it is high time to put this approach into practice in real-life case studies. Object-based distributed systems are an ideal target for this purpose, as they usually contain many components, either to model physically distributed entities or to represent concurrent activities taking place on a given execution site. This is the case with the ScalAgent deployment protocol, whose architecture consists of a tree of distributed agents, each agent being itself organized as a tree of concurrent activities.

From the verification activities undertaken in the Parfums project, we can draw a number of conclusions.

The complexity of the ScalAgent deployment protocol grows quickly as new components are added to the system. For instance, adding a new agent to deploy may start not less than 20 additional concurrent processes. For this reason, it seems that only compositional techniques have a chance to cope with the corresponding state explosion.

Because the ScalAgent deployment protocol is implemented in Java, we could have tried to apply a software model checker (such as the Java PathFinder [22] or Bandera [5]) directly on the Java source code. We did not choose such an approach because, to the best of our knowledge, these tools can only analyze programs running on a single Java

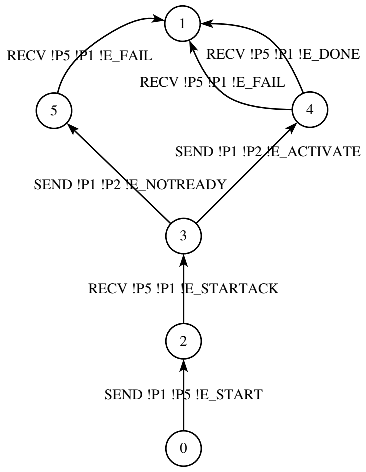


Figure 11: Service provided by the root controller, as an LTS obtained by compositional verification using CADP.

virtual machine (JVM), whereas the SCALAGENT protocol is designed for multiple machines, each running a separate JVM. We also felt that a process algebra such as LOTOS, with its built-in concurrency and abstraction primitives, would provide better support for compositional modeling and verification. As a consequence, our verification efforts mostly addressed the higher level design (i.e., the reference specification of the protocol) rather than the implementation (i.e., the JAVA code) although an examination of the latter was sometimes needed.

Technically, the results of the verification effort are positive. Several ambiguities were found in the reference specification and the verification work exhibited an undocumented assumption (synchrony of communications) of crucial importance for a proper functioning of the protocol. The use of compositional verification allowed to check significantly complex finite configurations within a reasonable amount of time (at the moment, configurations with 70 concurrent processes can be verified in less than 20 minutes).

In modeling the SCALAGENT deployment protocol, we chose to introduce many distributed buffers, instead of one central buffer. This avoids a bottleneck problem, which might prevent compositional verification from being applied [6]. We also took advantage of the "doubly nested" tree-like structure of the SCALAGENT deployment protocol. Originally designed to ensure the scalability of the protocol when deploying software components on many machines, this tree-like structure also forms the skeleton of our compositional verification scenarios, in which LTS generation and minimization phases are incrementally performed from the leaves to the root of the trees.

Last but not least, a complex system such as the ScalAgent deployment protocol could not be analyzed in absence of mature verification tools. We found the Cadp toolbox robust enough for this challenge, but had to extend it in several ways. Two existing tools (Exp.Open and Projector) had to be entirely rewritten for performance reasons. A new tool, Bcg_Graph, was introduced for fast, automatic generation of communication buffers. The SVL scripting language was enriched to allow a wider form of parameterization and to take advantage of Bcg_Graph functionalities. Interestingly, Svl scripts, originally to be written by human experts, are now automatically generated by the Ocaml translator. We observe here a situation in which new software layers are continuously added on top of existing ones, an integration trend that also occurs in other branches of computer science.

As regards future work, we foresee two directions. First, we are currently attacking larger configurations (90 and more concurrent processes) so as to discover the actual limits of compositional verification. Second, we seek to detect various livelock situations automatically using the Evaluator 3.0 model checker [18]; as the μ-calculus formulas needed to characterize livelocks may depend on the set of components defined in the Xolan architectural description, the Ocaml translator could be extended to generate these formulas automatically. This would reduce the risk of error and keep the verification fully transparent to the end-user.

Acknowledgements

The authors are grateful to Roland Balter, Luc Bellissart, and David Felliot, for sharing their knowledge of the ScalAgent deployment protocol. They would also like to thank the Parfums project manager, Laurent Coussedière, as well as David Champelovier, Radu Mateescu, and Wendelin Serwe for their useful comments about this report.

References

[1] Gilles Barthe, Dilian Gurov, and Marieke Huisman. Compositional Verification of Secure Applet Interactions. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering FASE'02 (Grenoble, France)*, number 2306 in Lecture Notes in Computer Science, pages 15–32. Springer Verlag, April 2002.

[2] Frances Brazier, Frank Cornelissen, Rune Gustavsson, Catholijn M. Jonker, Olle Lindeberg, Bianca Polak, and Jan Treur. Compositional Design and Verification of a Multi-Agent System for One-to-Many Negotiation. In *Proceedings of the Third International Conference on Multi-Agent Systems ICMAS'98*. IEEE Computer Society Press, 1998.

[3] 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.

[4] S. C. Cheung and J. Kramer. Checking Safety Properties Using Compositional Reachability Analysis. *ACM Transactions on Software Engineering and Methodology*, 8(1):49–78, January 1999.

[5] James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera: Extracting Finite-state Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering ICSE'2000 (Limerick Ireland)*, pages 439–448, June 2000.

[6] 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)*, pages 229–242. IFIP, Kluwer Academic Publishers, September 2001. Full version available as INRIA Research Report RR-4222.

[7] 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.

[8] Hubert Garavel and Frédéric Lang. SVL: a Scripting Language for Compositional Verification. In Myungchul Kim, Byoungmoon Chin, Sungwon Kang, and Danhyung Lee, editors, *Proceedings of the 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems FORTE'2001 (Cheju Island, Korea)*, pages 377–392. IFIP, Kluwer Academic Publishers, August 2001. Full version available as INRIA Research Report RR-4223.

[9] Hubert Garavel, Frédéric Lang, and Radu Mateescu. An Overview of CADP 2001. *European Association for Software Science and Technology (EASST) Newsletter*, 4:13–24, August 2002. Also available as INRIA Technical Report RT-0254 (December 2001).

[10] D. Giannakopoulou, J. Kramer, and S. C. Cheung. Analysing the behaviour of distributed systems using TRACTA. *Journal of Automated Software Engineering, Special issue on Automated Analysis of Software*, 6(1):7–35, January 1999.

[11] S. Graf, B. Steffen, and G. Lüttgen. Compositional Minimization of Finite State Systems using Interface Specifications. *Formal Aspects of Computation*, 8(5):607–616, September 1996.

[12] 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.

[13] 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 1989.

[14] Guoping Jia and Susanne Graf. Verification Experiments on the MASCARA Protocol. In Matthew Dwyer, editor, *Proceedings of the 8th International SPIN Workshop on Model Checking of Software SPIN'2001 (Toronto, Canada)*, volume 2057 of *Lecture Notes in Computer Science*, pages 123–142. Springer Verlag, May 2001.

[15] 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.

[16] Frédéric Lang. Compositional Verification using SVL Scripts. In Joost-Pieter Katoen and Perdita Stevens, editors, *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2002 (Grenoble, France)*, volume 2280 of *Lecture Notes in Computer Science*, pages 465–469. Springer Verlag, April 2002.

[17] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system (relase 3.06), documentation and user's manual, 2002. `http://caml.inria.fr/ocaml/htmlman/index.html`.

[18] Radu Mateescu and Mihaela Sighireanu. Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. *Science of Computer Programming*, 46(3):255–281, March 2003.

[19] Willem-Paul de Roever, Frank de Boer, Ulrich Hanneman, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency Verification – Introduction to Compositional and Noncompositional Methods*, volume 54 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, November 2001.

[20] 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.

[21] R. J. van Glabbeek and W. P. Weijland. Branching-Time and Abstraction in Bisimulation Semantics (extended abstract). CS R8911, Centrum voor Wiskunde en Informatica, Amsterdam, 1989. Also in proc. IFIP 11th World Computer Congress, San Francisco, 1989.

[22] W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In Yves Ledru, editor, *Proceedings of the 15th IEEE International Conference on Automated Software Engineering ASE'2000 (Grenoble, France)*, pages 3–12, September 2000.

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