INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Verifying Safety of Fault-Tolerant Distributed Components – Extended Version

Rabéa Ameur-Boulifa — Raluca Halalai — Ludovic Henrio — Eric Madelaine

N° 7717

Septembre 2011

Distributed Systems and Services

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--7717--FR+ENG

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE
INRIA
centre de recherche SOPHIA ANTIPOLIS - MÉDITERRANÉE

Verifying Safety of Fault-Tolerant Distributed Components – Extended Version

Rabéa Ameur-Boulifa* , Raluca Halalai† , Ludovic Henrio‡ , Eric Madelaine‡

Theme : Distributed Systems and Services
Équipe-Projet Oasis

Rapport de recherche n° 7717 — Septembre 2011 — 31 pages

Abstract: We shows how to ensure correctness and fault-tolerance of distributed components by behavioural specification. We specify a system combining a simple distributed component application and a fault-tolerance mechanism. We choose to encode the most general and the most demanding kind of faults, Byzantine failures, but only for some of the components of our system. With Byzantine failures a faulty process can have any behaviour, thus replication is the only convenient classical solution; this greatly increases the size of the system, and makes model-checking a challenge. Despite the simplicity of our application, full study of the overall behaviour of the combined system requires us putting together the specification for many features required by either the distributed application or the fault-tolerant protocol: our system encodes hierarchical component structure, asynchronous communication with futures, replication, group communication, an agreement protocol, and faulty components. The system we obtain is huge and we have proved its correctness by using at the same time data abstraction, compositional minimization, and distributed model-checking.

Key-words: Byzantine faults, distributed systems, software components, behavioural semantics, verification, model-checking, distributed model-checking

This work was partialy funded by the ANR international project ANR09-BLAN-0375-01 between INRIA and Un. of Tsinghua, Beijing, China.

* Institut Telecom, Telecom ParisTech, LTCI CNRS
† Technical University of Cluj-Napoca, Romania
‡ INRIA-I3S-CNRS, University of Nice Sophia Antipolis

Centre de recherche INRIA Sophia Antipolis – Méditerranée
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

Vérification de la correction de composants distribués tolérants aux pannes – Extended Version Étendue

Résumé : Nous montrons comment assurer la correction et la tolérance aux pannes de composants distribués à l'aide de spécifications comportementales. Nous spécifions un système combinant une application distribuée très simple avec un mécanisme de tolérance aux pannes. Nous avons choisi le type de fautes le plus général et le plus exigeant, les pannes Byzantines, mais seulement pour une partie des composants de notre système. Avec des pannes Byzantines un composant peut avoir n'importe quel comportement, et la replication est la seule solution classique convenable; ceci augmente de beaucoup la taille du système, et sa vérification par des techniques de model-checking est un défi. Malgré la simplicité de notre application, l' étude complète du système nous oblige à combiner de nombreux aspects nécessaires à l'application distibuée ou au protocole de tolérance aux pannes: notre système utilise une architecture de composants hiérarchiques, des communications asynchrones avec futurs, de la replication, de la communication de groupe, et un protocole de consensus. Le système obtenu est très gros, et nous avons pouvé sa correction en combinant des techniques d'abstraction de données, de minimisation compositionnelle, et de model-checking distribué.

Mots-clés : pannes Byzantines, systèmes répartis, composants logiciels, sémantique comportementale, vérification, model-checking, model-checking distribué

1 Introduction

Safety in distributed systems is a wide research area which needs to be tackled at several levels: from the safety of the execution platform, to the correctness of the communication protocols and to correctness of the distributed applications. This article aims at evaluating the adequacy of formal method techniques for the verification of real-size distributed applications. The objective tackled by this article is really challenging because the application we consider features several non-functional concerns which contribute to the explosion of the number of states that can be reached by the application. Indeed we choose to provide a model and prove properties for a distributed application featuring fault-tolerance similar to Byzantine fault tolerance (BFT).

Our work is placed in the context of component oriented programming. Indeed from a programming model point of view, components provide well-defined modularity, and easiness to compose large applications from the composition of basic blocks. Also components require the precise definition of interfaces through which the basic blocks cooperate, which is crucial for a precise design of an application, but also strongly helps the formal specification of the application. Our components also allow a hierarchical and modular design, better specifying the structure of the application. We choose GCM[2] as our component model because it is naturally adapted to distribution, hierarchy, and one-to-many communication, but also it provides reconfiguration capabilities which we want to consider in future works. GCM is an extension of the Fractal component models with support for deployment, scalability, autonomic behaviour, and asynchronous communication; it also shares a lot of similarities with SCA [3]. In the VerCors [8] platform, we provide tools for verifying the behaviour of such distributed component applications.

This paper shows how to specify the behaviour and to verify properties of distributed component applications with request queues, future proxies and group proxies, and one-to-many interfaces. To illustrate our approach, we choose a simple distributed application featuring fault-tolerance by replication. Though the fault-tolerance properties we address are not outstanding, we think this application is a good opportunity to investigate on the use of model-checking to ensure safety of fault-tolerant applications. This article has the following objectives:

- Promote the use of formal methods to ensure safety of distributed systems.
- Provide a model for one-to-many communication.
- Study the modelling of faulty processes, and investigate the use of model-checking for verifying fault-tolerance from an application point of view. Indeed, most of the existing studies on this domain focus on the proof of correctness of the protocols only, not on the whole distributed application [14].
- Investigate the adequacy of distributed model-checking for verifying a distributed and asynchronous application that generates a huge state-space.

We do not model reconfiguration and adaptation, but we design our specification in such a way that those aspects can be added to the model in the future.

In the following, Section 2 presents the related works, with a particular focus on BFT and GCM components. Then, we describe our fault-tolerant application

and its modelling in Section 3. Finally, Section 4 describes the distributed model checking phase and the properties we verify.

2 Background and Related Works

2.1 Formal Methods for Component Models

As the formal methods matured, they have been integrated into environments that support the development of component-based systems. They ensure the correct behaviour of the assembly of complex applications in all the stages of the development lifecycle (from specification to execution). However, although those frameworks share the same basic concepts, they substantially differ in the range of application domains and supported features. For instance, some of them are dedicated to embedded systems verification [10, 4] while the others are dedicated to software engineering. We focus below on related works for behavioural specification and verification of distributed components.

Creol [19] is a programming model featuring active objects, requests and futures, similarly to our approach. A framework provides component modelling for Creol; it provides a formal language [13] that supports compositional reasoning and makes automatic testing and verification possible. This language is defined over communication labels, and specifies components in terms of traces of observable behaviour at the interfaces.

Cadena [16] is an environment for modelling and verifying CCM component-based systems. The framework offers a rigorous type-based language [20] for describing component connectors, and the interaction between them. The compositional analysis is based on the assume-guarantee reasoning. However, the component model does not support hierarchical structure.

SOFA [24] is a framework for developing distributed systems. It supports component-based development as well as formal verification. The SOFA 2 component model is hierarchical and supports reconfiguration, making it quite close to ProActive/GCM even though one-to-many communication and asynchrony with futures are not offered by default in SOFA. SOFA uses "behaviour protocols" for specifying possible interactions between components and checking the correctness of the assembly, making the verification process in SOFA quite different from ours, but our approach could also be applied to SOFA components.

This article relies on the pNets [1] formalism for describing the behaviour of parametrized networks of LTSs. We showed in [1] how to build models for GCM components, asynchronous communication, and futures. [7] describes how to specify group communication in pNets. Additionally to faulty components, this article extends the preceding semantics by specifying one-to-many communication at the GCM level, and the management of proxy instances.

The CADP toolset [11] is one of the prominent platforms for the specification, verification, and testing of distributed systems in the academic landscape. It handles several input formalisms, and provides an extensible API. The toolset includes engines for building hierarchically the state-space of systems, building and manipulating LTSs on distributed infrastructures, minimizing LTSs along several behavioural equivalences, model-checking properties, checking equivalences between systems, building test suites, evaluating performances, etc.

2.2 Verifying Byzantine Fault-tolerant Systems

Byzantine fault tolerance (BFT) has a long history [22, 26]; results in this research area are very difficult to obtain and to prove. Indeed, BFT supposes that a faulty process can have any behaviour. The name BFT comes from the original problem raised by Lamport relying on Byzantine generals that must all take the same decision (attack or retreat), knowing that some of the generals are traitors. Traitors can say anything to the others, but the others must all act identically. In computer science, this situation represents either a faulty process behaving "randomly" or a malicious entity. BFT has gain new interests since the apparition of a new form of large scale distributed computations relying on entities that, by nature, cannot be trusted. Typically a P2P storage application cannot make any assumption on the kind of misbehaviour the peers can have.

The purpose of this paper is not to prove that a BFT protocol is correct but to understand whether it is possible to represent all the aspects of a complete component application communicating by request-replies, and at the same time reason about the fault-tolerance of this entire application. We focus on a specific application similar to [21] but simplify it: our application consists of a Master component replicating data to be stored on several workers. The master updates the worker value, and gathers replies from workers to retrieve the stored value. If enough non-faulty workers are instantiated, and enough identical replies are returned to the master, the stored value can be retrieved. The objective of this paper is not to study the implementation of the component model, this is why we make the assumption that communications are performed safely. More precisely, we suppose that the middleware ensures that messages systematically follow the bindings, and that a component can only reply to the requests it received. For example, a faulty component cannot communicate to any component of the application, and a faulty components cannot reply instead of a non-faulty one.

Note that the master is supposed to be non-faulty; Protocols for dealing with a faulty master exist and have been heavily studied and implemented. For example, recently [21] implemented a BFT storage in the same settings as our application. Here we simplify the problem and focus on the correct handling of faulty workers, similarly to the case studied in Section 4.2 in [26]. If f is the number of tolerated faults, $2f + 1$ slaves are sufficient for reaching a consensus. However, as it is generally required in BFT, i.e. when the master can be faulty, we instantiate $3f + 1$ slaves. Section 4 will show that specifying a whole application with those simplifying hypotheses already requires the full power of distributed model-checking over a cloud-like architecture.

Our approach for encoding Byzantine faults is the following: faulty slaves can feature any behaviour, upon verification the model-checker will then explore all the possible behaviours, including the malicious ones. We then specify a simple agreement procedure where the Master component waits until enough slaves answered correctly. In order to count them, our architecture description is aware of which slave is faulty, but the business code does not use this information.

2.3 Distributed Components and their Semantics

This section recalls the component structure and semantics of GCM, a complete definition can be found in [17].

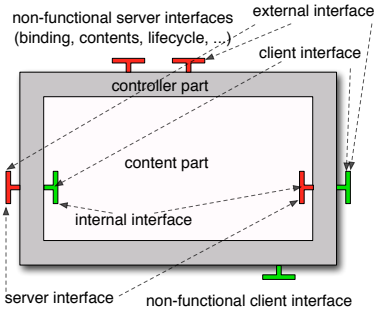

Figure 1: A GCM component

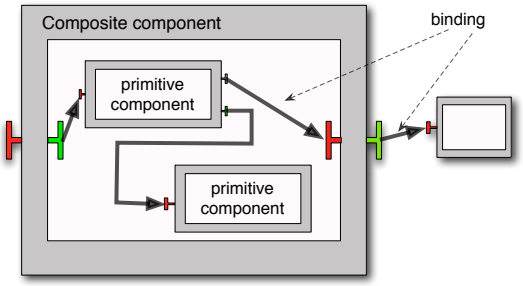

Figure 2: A component system

Component Structure. The structure of GCM components is inherited from Fractal: A GCM component can be either *composite* (i.e. composed of subcomponents), or *primitive* (a basic element encapsulating the business code). A component comprises a content (providing the functional code) and a membrane (a container managing non-functional operations). The interfaces are the only access points to components. Each interface is either *client* (emitting invocations) or *server* (receiving invocations). We distinguish *functional* interfaces addressing the business of the application from *non-functional* ones invoked to manage, monitor, and introspect the application. A *binding* connects a client interface to a server interface (Fig. 2); a message emitted by a client interface is transmitted to the server interface bound to it. In composite components, interfaces are either *internal* – exposed to the subcomponents – or *external* – exposed to other components. The interface cardinality indicates how many bindings can be made from or to this interface. In this paper, we only use two interface cardinalities: singleton (one-to-one binding) and multicast (one-to-many binding). The different parts of a GCM component are shown in Fig. 1, whereas Fig. 2 shows an assembly of components bound together, on the left there is a composite composed of two primitives; the figure also illustrates different bindings.

Communication. The basic communication paradigm in GCM is asynchronous message sending: communication consists in synchronously dropping a message in a *request queue* at the receiver side, and creating a future to represent the result of the invocation. A future is an empty object representing the result of a computation performed in parallel. Once the future is created, the execution continues immediately on the sender side. When the request treatment is finished, the result is automatically returned to replace all the references to the corresponding future. When a component accesses a future, it is blocked until the result is returned. However, future references can safely be passed between components, inside invocation parameters, or inside a request result. To prevent shared memory between components, parameters and results are copied; no object is passed by reference.

A *multicast* interface is a client interface that transforms a single invocation into a list of invocations, sent in parallel to a set of connected interfaces. The result of an invocation on a multicast interface is a list of results. Invocation parameters can be distributed according to a distribution policy that can be customized. Typical distribution policies include *broadcast* that sends the same parameter to each connected component, and *scatter* that splits the parameter.

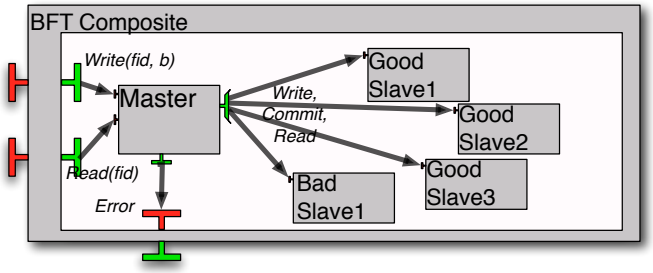

Figure 3: Component Structure of our application

Component Behaviour. Primitive components encapsulate the business code, their behaviour is highly dependent on the application; it is provided by the application programmer. The only constraints they must respect are: they serve requests of the request queue, they emit new requests on their client interfaces, and can receive a result for the futures they hold. We consider here only mono-threaded primitive components: a single request is served at a time.

By contrast, composite components have a predefined behaviour: they serve requests in the reception order, and delegate the requests to sub-components, according to the bindings. For example, when a composite component receives a request from the outside, it delegates its service to one of the sub-components.

3 Our Fault-tolerant Application and its Specification

This section describes informally our application, and then presents its behavioural model. We present the architecture using the pNets model [1], a formalism to encode labeled transition systems with value passing, parametrized topologies of processes, and different types of communication. We describe then the primitive component internal behaviour, and the semantic-level process generated from the GCM architecture. We focus on the parts of the specification that are directly related to one-to-many communications and fault-tolerance, details of the other processes are given in [6].

3.1 Distributed Component for Fault-tolerant Storage

Fig. 3 shows the architecture of our application. It consists of a main composite component BFT-Composite. The white part of the composite is the functional content made of a Master component and several slaves. Some of those slaves are called good slaves, i.e. non-faulty, the bad ones are faulty and behave randomly. In practice one never knows which of the slaves is good or bad but it is necessary that the verification process knows this information to be able to count the number of good and bad slaves.

Properties of Interest. From a high-level point of view, we are interested in the storage properties of our application: *the stored value can be retrieved unchanged, even if some of the slaves are faulty.* Of course, some additional properties are crucial like: the master always finally answers to the requests it receives. Also, the master must rely on the slaves for storing the value, and does

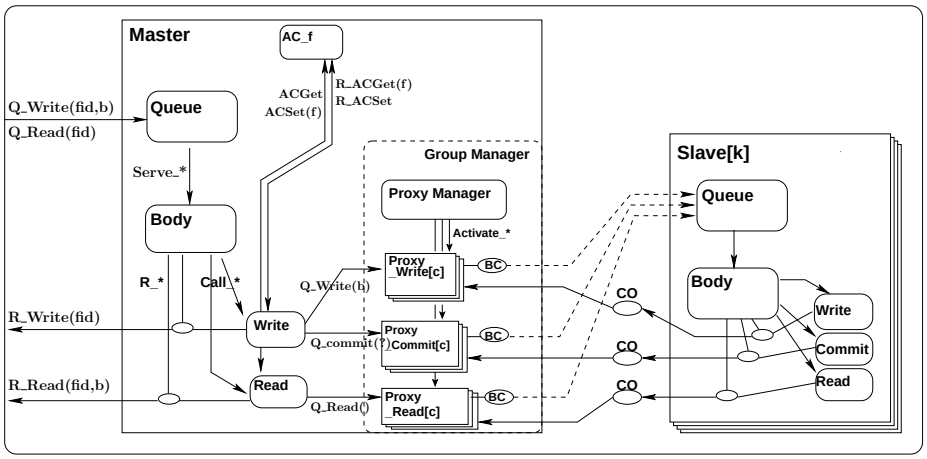

Figure 4: pNet Architecture for the whole system

not distinguish good slaves from bad slaves, for example, for writing data the master must broadcast a write request to all the slaves.

3.2 Architecture

We describe here the architecture of the semantic model of our use-case. The overall architecture of the system is shown in Fig. 4. It is composed of:

- An indexed family of slaves receiving invocations from the master. Each of them has a queue[1] storing the requests not treated yet, a body part describing how to treat the incoming requests and delegate them to the behavioural specification of methods Write, Commit, and Read. Each requests can reply to the master by updating a future (represented by the arrows between the Write box and the CO element). The system is instantiated with 3 good slaves and 1 bad slave.

- A Master component receiving requests from a client and forwarding them to the slaves (that are bound to it). It also has a request queue and a body delegating the treatment of requests to sub-parts of the master. Treatment of read and write methods will be detailed below.

- The connections that are one-to-one bindings, except for BC (broadcast) that dispatches a request from the master to all the slaves it is bound to, and CO (collect) that carries a reply from one of the slaves to the appropriate proxy. Those 2 bindings will be detailed in Section 3.3.1.

To optimize the size of the model, the composite has no request queue and calls are directly issued to the *Master* component. This has no consequence because the requests are directly delegated to the Master component, and the request queue of the Master is sufficient for dealing with asynchrony.

[1] We generate the behaviour of each request queue as an individual process able to store a finite number of requests with their parameters

3.3 The Master Body and its methods

Let us first describe the communication patterns and name conventions that we use in this paper. All local methods are triggered by a first outgoing communication of the form `!Method`, then the response is received as parameter of a `?R_Method` incoming communication. For example, in Fig. 6 `!Get_Write_Proxy` requires a new group proxy for invoking the Write method on the slaves. The proxy is returned and stored into `p1` by the reply: `?R_Get_Write_Proxy(p1)`. On the other side, method invocation towards remote components are of the form `!Call_Method`, those method invocations enqueue a request in the remote request queue, and pass a proxy reference as one of the parameters of the invocation. The remote method will, upon termination, fill the proxy with the calculated value; for this, the `!R_Method` transition synchronizes at the same time with the invoker that receives the value and with the body of the component containing the method, so that next request can be served.

The master body. The body is encoded in generic way: it serves sequentially functional and non-functional requests. In this work, we only use the service of each functional request (on method Read, Write, or SetF). This service calls the adequate method (e.g., `!Call_Read`), and waits until the method terminates, signaled by `R_` events (e.g., `?R_Read`); `R_Read` synchronizes both with the component that triggered the request and with the body. As requests are served one after the other, this encodes a mono-threaded behaviour for the master.

The Attribute controller. In Fractal, the attribute controller provides read and write access to the attributes of the components; the only attribute of the Master component is `f` – the number of faults that can be handled. The behaviour of the attribute controller is very simple: it simply provides a setter (`ACSet`) and a getter (`ACGet`) method for storing and retrieving the value of `f`.

The Collate method. Based on the vector of replies received by the proxy, this method computes a consensus in order to know whether enough slaves returned a correct answer. It is used by the methods Read and Write described below.

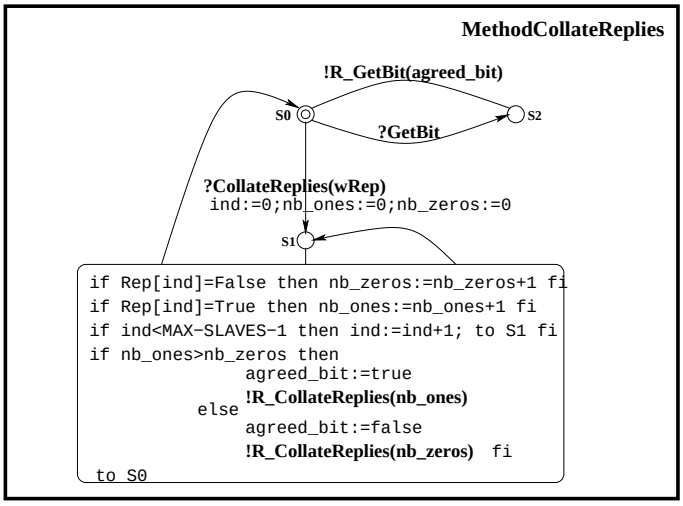

Figure 5: Behaviour of the method: MasterCollateReplies

Fig. 5 represents the behaviour of Collate in a format similar to Statecharts [15]: starting from initial state `S0`, Collate is always used by first triggering a `?CollateReplies` sending it a vector of replies currently known; then from state `S1`, a complex transition counts the number of `True` and `False` in the vector. It stores in `agreed_bit` the reply the most frequent and returns (by `!R_CollateReplies`) the number of replies that agreed on this value. Then, the agreed value can be retrieved by a `?GetBit`, that returns the `agreed_bit` value.

The Write method. The write method is the most complex method of our example, it is shown in Fig. 6. It first gets the current value of `f`, read from the attribute controller, and initializes the variables `agree`, `awaited`, and `nb_Slave`. It consists of two phases; first, a write request is sent to all the slaves, then the master waits until enough slaves agree on the reply, `agree` is the number of necessary identical replies, and `awaited` is the number of awaited replies. If necessary, additional replies are awaited, and `awaited` is incremented. It is not possible to wait for more replies than the number of slaves; if such a situation occurs, it means that the BFT hypothesis is not verified, more exactly, more than `f` slaves are faulty and an error is raised. When enough identical replies have been received, the write method enters a commit phase that behaves similarly to the write phase. At the end the method returns to the initial phase, emitting a `!R_Write` that also indicates the end of the method.

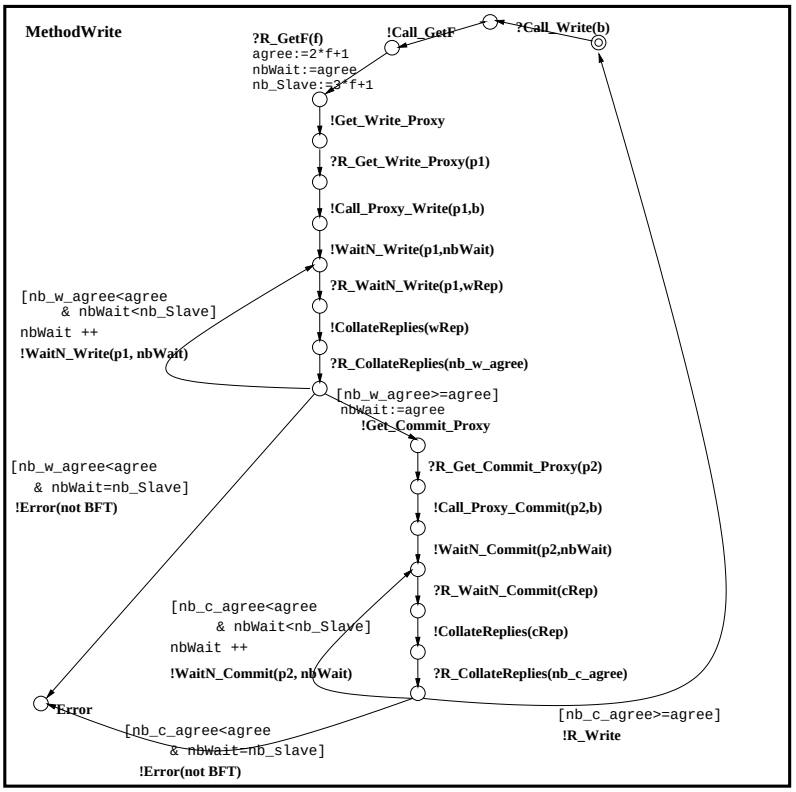

Figure 6: Behaviour of the Write method

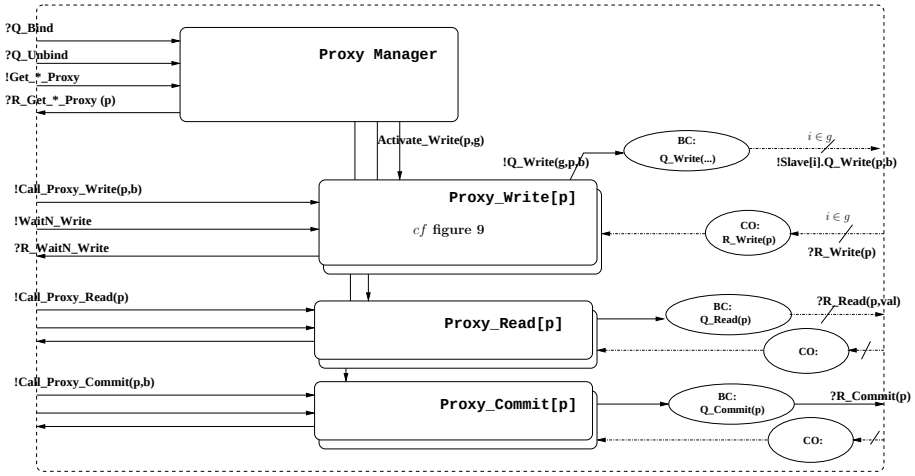

Figure 7: Focus on the elements for managing the group

The Read method. The behaviour of the Read method is very similar to the Write method above. The main difference is that, after triggering remote invocations and waiting for enough identical replies, it inputs the agreed bit found by the collate method and returns this value to the client.

3.3.1 The Master Proxies

Managing groups of slaves. We first focus on the management of groups of slaves, i.e. groups to which the write, read and commit requests will be addressed. The part of the pNets that deal with this aspect is shown in Fig. 7. It includes a proxy manager (Fig. 8) that returns an available proxy through its `Get_*_Proxy` invocations. If reconfiguration was enabled, it would receive bind and unbind requests for adding or removing slaves. When a new proxy is requested, one proxy is activated (among the families of `Proxy_write`, `Proxy_Read`, or `Proxy_Commit` proxies), and given the group `g` on which next invocation will be performed. A reference to this proxy is returned, and can be used to remotely invoke Write, Read, or Commit on the slaves. The group `g` passed upon activation is used later inside the broadcast communication: the circle `BC: Q_Write(...)` performs a synchronization involving the proxy and all the slaves of `g` sending them the same invocation, `!Slave[i].Q_Write(p,b)`, where `p` is the proxy identifier. The symmetric communication is performed by the `CO: R_Write(p)` that collects replies from all the slaves of `g` and returns them to the `Proxy_Write` pNet: each member of `g` can send a reply to the master. Note that `g` can be modified inside the manager and a copy of the group is passed upon activation of a proxy. This guarantees that the CO operation will be performed on the same group as the invocation, even if, in the manager, the group is changed in the meantime.

The Write proxy. (see Fig. 9) Upon activation, the write proxy waits for an invocation from the master write method. It then initializes the `WRep` array of received replies as well as `len` – the number of replies currently received. Its

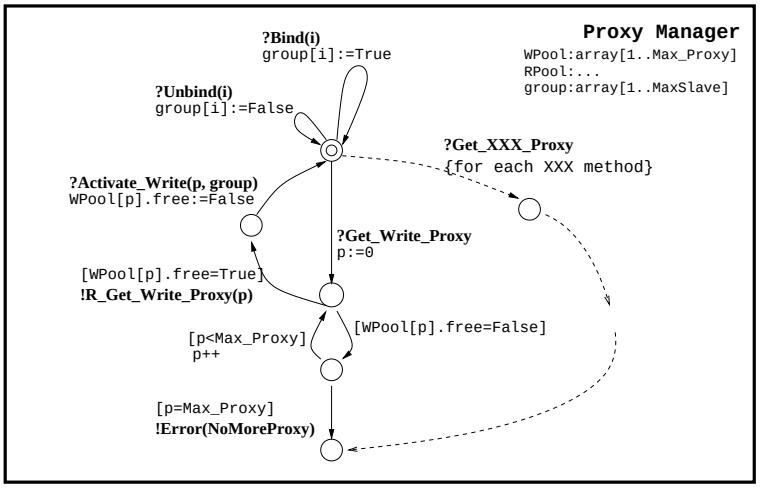

Figure 8: Behaviour of the Proxy Manager

two main behaviours are then (1) to receive a reply from an element of the group, which updates the `Wrep` array, and the `len` value; and (2) to fulfill a `WaitN_Write` invocation from the master write, which returns the current array of received replies once the number of awaited replies is reached. Proxies for read and commit method are similar to the write request proxy.

3.4 The Slave Components and their methods

The behaviour of the slaves is much simpler than the one of the master. We encode two kinds of slaves: good slaves behave as expected, whether bad slaves behave randomly and encode the byzantine faulty processes. We instantiate as many faulty processes as the number of faults we can tolerate. The fact that the system description distinguishes between faulty and non-faulty processes has no influence here because the functional parts of the components never use this knowledge: the code of the Master component never distinguishes between the communications towards the faulty slaves, and towards the non-faulty ones.

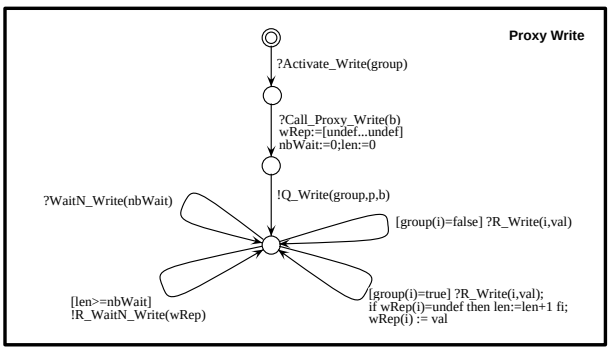

Figure 9: Behaviour of the Write request proxy (Proxy for Read and Commit are similar)

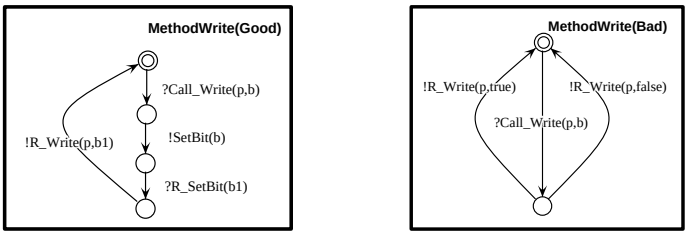

Figure 10: The Write method of the (Good and Bad) slaves

The slave body serves successively the requests (Commit, Read and Write) arriving at the slave queue much similarly to the master body. The bad slaves and the good slaves have the same body, they all serve the request in a FIFO order, and no two requests are served at the same time: the slaves are mono-threaded. The slaves have three methods: Write, Read and Commit; we show the method `Write` for the good and bad slaves in Fig. 10, the behaviour of a good slave consists in storing the bit value `b` received thanks to a call to `!SetBit` that sets a local attribute of the slave. There is a method `!GetBit` for reading this value, it is called by the `Read` request. The bad slave as shown in Fig. 10 replies randomly to each individual request. The commit phase is here to show how a commit phase would be implemented, but it is not used by our slaves: it would be useful if the master could also have a faulty behaviour.

According to the BFT hypothesis, a bad slave can behave arbitrarily. However, we have to restrict a little this behaviour so that it can be encoded and verified by finite model-checking techniques. Here are the hypotheses we make and the reasons why it is safe to make them:

- Bad slaves do not steal the identity of another entity: we suppose here that the underlying middleware guarantees the identity of the components sending requests or replies. It is the classical "oral messages" assumption of [22].
- Bad slaves only reply to required requests. We suppose again that the middleware verifies this to guarantee the integrity of the program execution.
- Bad slaves only reply to requests in the order required. This assumption is stronger but we can show that it has no influence on the final result. First, the master is single threaded, waits for enough replies before requiring another computation, and does not access the future afterward; thus late replies would have no influence on the computation. In principle, a bad slave could serve the request in the wrong order and use this information to behave in a malicious manner; but the exhaustive exploration of all the possible replies is even more general than the scenarios using out of order service of requests.

4 Building the model, and running the verification tools

In this section we describe the methods and tools used to build the behavioural model of our application and to check its properties, and we discuss the combination of advanced techniques we have used to master the model complexity.

We build the behavioural model of our case-study in three steps (Fig. 11). From the specification of the component architecture and behaviour, our tool ADL2N [8] builds a hierarchical and parametrized pNets model, including the data types, the behaviour, and the architecture of the system. Then abstractions are applied on the data domains, yielding a finitary model. Finally the model is encoded using a combination of several input formalisms from the CADP toolset [11]: the Fiacre language [5] provides syntax for data types and expressions, definition of LTS, and a form of composition of processes by synchronization on channels; the EXP and SVL languages [11] support the hierarchical encoding of our pNets, and the scripting of the various verification tasks.

Then we run a combination of CADP tools, the most important ones are: ceasar.open for generating transition systems from Fiacre programs, either on a single machine, or on parallel infrastructures when used in combination with distributor; exp.open to build product of transition systems described in EXP format; and Evaluator4, the new version of the model-checker that deals with the MCL (Model Checking Language) logics [23], which is an extension of the alternation-free regular μ-calculus with facilities for manipulating data.

The Vercors[2] tool platform should assist the programmer in the encoding and verification of his application. It includes the Vercors editors, the ADL2N, ABS and N2F tools; it is currently under development. For this paper, we already have been able to generate approximately 50% of the Fiacre and EXP code.

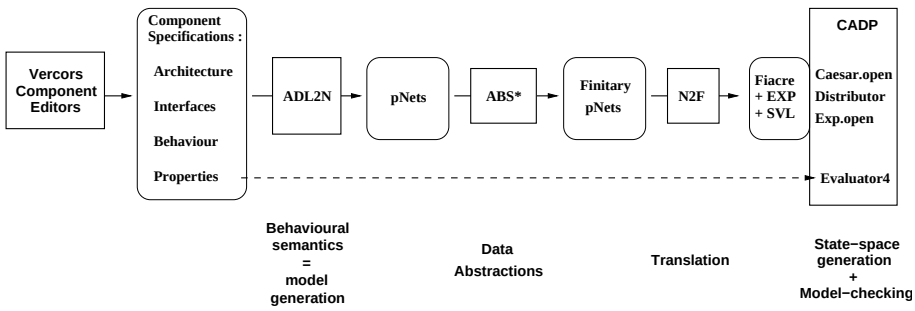

Figure 11: Tool chain and corresponding processing steps

One goal of this work is to experiment with various methods for mastering the state explosion inherent to large models, such methods consist of:

1. data abstraction
2. hierarchical hiding and minimization
3. use of contextual environment information
4. distributed state-space generation

[2] http://www-sop.inria.fr/oasis/index.php?page=vercors

We have used 1) in several ways. First, all data variables have been given abstract types with (very small) finite domains, in fact we choose the smallest abstract domain that preserves the formulas to be proven. Secondly, the topology parameters of the system (the number of slaves and number of proxy instances) have been reduced to a minimum number, though significant for our scenario; proving properties that would be valid for any values of such parameters is out of the scope of model-checking. Finally, the request queues raises another issue: their explicit representation has a size exponential in the number of values that the queue cells admit. Our approach is to encode a (small) finite model of the queue, including events denoting an error when this finite queue is out-of-bounds. Then we check by model-checking whether this event is reachable, or the chosen size is sufficient. The soundness of these approaches is worth discussing; for the domains of value-passing parameters, we can define finite abstractions that preserve safety and liveness properties [9]; for the length of queues, we are building an under-approximation, and we check explicitly its validity. But for topology parameters, we have no such general result and we only prove properties for a given instantiation, that is already very helpful as a "debugging tool". Proving more general properties is not in the scope of this paper.

Method 2) is now quite classical when using bisimulation-based tools. Let us remark that to be optimal, we have to generate models specifically for each formula to prove. Method 3) has been proposed and advocated by the CADP developers, and is indeed very important when combined with 2). The problem arises when you build subsystems hierarchically without taking into account the specific way in which other pieces of the system interact with a given subsystem. The context information can be built automatically by the CADP tools from the behaviour of the other subsystems (in which case it is guaranteed to be sound), or can be specified manually (that may lead to under-approximations). We chose the second option, and we used the context behaviour to reduce further the possible values of input data of some methods, by symmetry arguments.

Method 4) is a hot research topic. We are using a local Cloud platform, providing large computing resources (>1300 cores and 3 Tbytes of RAM), where we can submit jobs in the form of task workflows. In our case, tasks consist of compilation of input formalisms, generation of transition systems for subsystems, minimization and product of systems, and model-checking. Tasks can be parallel, but for the current version of CADP, only LTS generation can run in a distributed way [12]. We were able to build systems with more than 10^9 states explicitly stored in distributed memory [18], but then the bottleneck is the merging of this structure before minimization or model-checking on a single machine. In practice, the good strategy is to decompose the system in such a way that subsystems are of reasonable size, or can be strongly constrained by contextual information, and to run concurrently the tasks computing the behaviour of each subsystem. Then minimization, product, and model-checking tasks are run as soon as their inputs are available, in a coarse-grain concurrent workflow.

Parameter Domains and System Sizes. We ran the use-case with 3 good slaves and 1 bad slave, allowing for 1 failure. We also generated the model in two different configurations, with the length `Q` of the Master Queue respectively 2 (for OutOfBounds detection) and 1 (for optimization).

As we do not have yet enough tool support at the level of the formalism compilers, we had to do a significant part of the Fiacre/Exp/SVL programming by hand, so we chose to build one single model with enough events visible to prove our formulas of interest. The intermediate code consists in 43 Fiacre processes for a total of 2900 lines of code, and of 330 lines of synchronization vectors in EXP format encoding 240 pNet structures.

Then the system is divided in 12 subsystems (9 for the Master itself); each part is encoded in a Fiacre source file, and its state space computed using distributor. So we have at this level 12 independent tasks in our workflow, running on 2 to 10 cloud nodes each. Each resulting automaton is reduced by branching bisimulation (with as much local actions hidden as possible), before being composed in a hierarchical way, using 4 synchronization products. The final product is minimized again, before running Evaluator4 for checking our properties. Decomposing the system in an efficiently manner currently requires human operation: the choice of subsystems is a compromise between: identifying processes that may be reused easily (through relabeling); defining subsystems that are big enough to take advantage of a distributed generation; choosing pieces which environment behaviour is well-specified.

The system sizes (states/transitions, after minimization) and computation times are summarized in the following table:

Q. size	Queue	Intermediate	Master	GoodSlave	Global	Total time
Q=1	21/229	542/3107	2M/45M	744/6550	22K/110K	10’
Q=2	237/3189	542/3107	5.8M/103M	5936/61K	34K/164K	59’

The middle columns in the table give reduced sizes for the most interesting subsystems: the Master queue, the biggest intermediate subsystem in our decomposition of the Master, the whole Master component, the (good) Slave component, and finally the global system, comprising the Master, 4 Slaves, and a Client. The last column gives the global computation time.

Correctness Properties. Once the behavioural model generated, we verified several properties, written using the MCL logics; they express various facets of the system correctness. Some properties express global correctness of the application, seen from the (external) client point of view. Others require the visibility of some internal events of the system, and reveal the feasibility of several scenarios, or the impossibility of some errors.

Let us start with simple reachability properties: all requests (Write or Read) sent to the system can terminate and return successfully. The first formula means that for each possible value of fid (the identifier of a client request), the action R_Read denoting the return of the corresponding Read request is reachable with some returned value val. This property is **True**, meaning that the Read request can terminate (this holds also for Write requests).

```
forall fid:nat among {0...2}. exists b:bool.
       <true* . {R_Read !fid !b}> true
```

Next formula checks the reachability of the BFT Error events. This property is **False**, meaning that we instantiated enough good slaves.

```
< true* . 'Error (NotBFT)'> true
```

We then ensure that the Master's queue cannot receive too many requests. Its validity depends on the system client(s). Here we have proved that a queue depth of 1 is sufficient to prove all of our correctness properties, if we have a single client, and if this client waits for replies before sending the next request.

```
< true* . 'Error (Master-OutOfBounds)'> true
```

Also, we have proved *Inevitability* properties like the following one. It ensures that it is (fairly) inevitable that after a Write request, either the system sends the corresponding Write response or raises an error. Here fairness means "fair reachability of predicates" in the sense of Queille and Sifakis [25]:

```
[ true* . ({Q_Write ?fid:nat ?bit:bool})
        . (not ('Error.*' or {R_Write !fid}))* ]
< (not ('Error.*' or {R_Write !fid}))*
     . ('Error.*' or {R_Write !fid}) > true
```

Similarily, we have shown that it is fairly inevitable that Read requests are replied, and also that the system is functionally correct: after a Write request (and before the next one), a Read request will answer with the correct value.

To summarise, we proved by model-checking that our application consisting of 1 master and 4 slaves (3 good ones and bad one) behaves correctly: 1) it answers to Read and Write requests, 2) the answers are correct in the sense that the read value is the value that has been written, 3) for this it relies on the slaves for storing the data (the master only performs a consensus), and 4) enough good slaves have been instantiated and the `NotBFT` error cannot be raised.

5 Conclusion

This paper shows the modelisation and verification by model-checking of a system that features: one-to-many communication, asynchronous communication with futures, byzantine faults, replication, and consensus. We showed here the possibility to encode and verify the correct behaviour of a whole distributed application that tolerates some faulty processes. Handling byzantine faults is a difficult task, because no assumption can be made on the behaviour of the faulty processes. Such a random behaviour makes automatic verification of the correction of a whole application even more difficult because a lot of possible states must be considered.

A next step could be to integrate the generation of faulty process, replication management, and consensus methods to our specification environment: the user would identify the possibly faulty components and the environment would generate BFT-like behaviour and replication for those components, but also broadcast and consensus operations. The new system could then be model-checked to decide whether the whole application is fault-tolerant.

Another lesson drawn here is that the behaviour of the whole application is huge, we used all the power of the distributed version of CADP on a cloud-like environment to verify the application. This shows that application-level fault-tolerance can be verified by a model-checker, but also that adding any other feature to the system (e.g. reconfiguration for changing the number of replicates at runtime) may be very difficult. To master such complexity we should use semantic properties of the programming model and of the middleware to get better and smaller abstractions at the level of the generated behaviour.

References

[1] T. Barros, R. Ameur-Boulifa, A. Cansado, L. Henrio, and E. Madelaine. Behavioural models for distributed Fractal components. *Annals of Télécommunications*, 64(1-2):25–43, 2009.

[2] F. Baude, D. Caromel, C. Dalmasso, M. Danelutto, V. Getov, L. Henrio, and Ch. Pérez. GCM: a grid extension to Fractal for autonomous distributed components. *Annals of Télécommunications*, 2009.

[3] M. Beisiegel, H. Blohm, D. Booz, M. Edwards, and O. Hurley. SCA service component architecture, assembly model specification. Technical report, March 2007.

[4] S. Bensalem, M. Bozga, T.-H. Nguyen, and J. Sifakis. Compositional verification for component-based systems and application. *IET Software*, 4(3), 2010.

[5] B. Berthomieu, J.P. Bodeveix, M. Filali, H. Garavel, F. La ng, F. Peres, R. Saad, J. Stoecker, and F. Vernadat. The syntax and semantics of Fiacre. In *Rapport LAAS #07264 Rapport de Contrat Projet OpenEmbeDD*, Mai 2007.

[6] R. Ameur Boulifa, R. Halalai, L. Henrio, and E. Madelaine. Verifying safety of fault-tolerant distributed components (extended version). Research Report RR-7717, INRIA, August 2011.

[7] R. Ameur Boulifa, L. Henrio, and E. Madelaine. Behavioural models for group communications. In *WCSI-10: International Workshop on Component and Service Interoperability*, Malaga, Spain, 2010.

[8] A. Cansado and E. Madelaine. Specification and verification for grid component-based applications: From models to tools. In F. de Boer, M. Bonsangue, and E. Madelaine, editors, *FMCO'08*, volume 5751 of *LNCS*, pages 180–203. Springer, Heidelberg, 2008.

[9] R. Cleaveland and J. Riely. Testing-based abstractions for value-passing systems. In J. Parrow B. Jonsson, editor, *Int. Conf. on Concurrency Theory (CONCUR'94)*, volume 836 of *LNCS*, pages 417–432. Springer, Heidelberg, 1994.

[10] J. Eker, J. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Sachs, and Y. Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, January 2003.

[11] H. Garavel, F. Lang, R. Mateescu, and W. Serve. Cadp 2010: A toolbox for the construction and analysis of distributed processes. In *TACAS'11*, volume 6605 of *LNCS*, Saarbrücken, Germany, 2011. Springer, Heidelberg.

[12] H. Garavel, R. Mateescu, D. Bergamini, A. Curic, N. Descoubes, C. Joubert, I. Smarandache-Sturm, and G. Stragier. Distributor and bcg_merge: Tools for distributed explicit state space generation. In J. Palsberg H. Hermanns, editor, *TACAS'06*, volume 3920 of *LNCS*, pages 445–449. Springer, Heidelberg, 2006.

[13] I. Grabe, M. Steffen, and A. B. Torjusen. Executable Interface Specifications for Testing Asynchronous Creol Components. Research Report 375, University of Oslo, Dept. of Computer Science, July 2008.

[14] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. The next 700 BFT protocols. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 363–376, New York, NY, USA, 2010. ACM.

[15] D. Harel. Statecharts: A visual formalism for complex systems, 1987.

[16] J. Hatcliff, W. Deng, M. B. Dwyer, G. Jung, and V. Ranganath. Cadena: An integrated development, analysis, and verification environment for component-based systems. In *Proc. of the 25th Int. Conf. on Software Engineering*, 2003.

[17] L. Henrio, F. Kammüller, and M. Rivera. An asynchronous distributed component model and its semantics. In F. de Boer, M. Bonsangue, and E. Madelaine, editors, *FMCO'08*, volume 5751 of *LNCS*, pages 159–179. Springer, Heidelberg, 2008.

[18] L. Henrio and E. Madelaine. Experiments with distributed model-checking of group-based applications. In *Sophia-Antipolis Formal Analysis Workshop*, page 3p., France Sophia-Antipolis, Oct 2010.

[19] E. B. Johnsen, O. Owe, and I. C. Yu. Creol: a types-safe object-oriented model for distributed concurrent systems. *Journal of Theoretical Computer Science*, 365(1 – 2):23 – 66, 2006.

[20] G. Jung and J. Hatcliff. A type-centric framework for specifying heterogeneous, large-scale, component-oriented, architectures. *Science of Computer Programming*, 75(7):615–637, 2010.

[21] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 45–58, New York, NY, USA, 2007. ACM.

[22] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4:382–401, July 1982.

[23] R. Mateescu and D. Thivolle. A model checking language for concurrent value-passing systems. In K. Sere J. Cuellar, T. S. E. Maibaum, editor, *FM'08*, volume 5014 of *LNCS*. Springer, Heidelberg, 2008.

[24] P. Parizek and F. Plasil. Assume-guarantee verification of software components in sofa 2 framework. *Software, IET*, 4(3):210 –211, june 2010.

[25] J.-P. Queille and J. Sifakis. Fairness and Related Properties in Transition Systems — A Temporal Logic to Deal with Fairness. *Acta Informatica*, 19:195–220, 1983.

[26] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22:299–319, December 1990.

A Behaviour of Other Methods

We include in this appendix the behaviour of the methods we could not fit in the body of the paper; we will publish a research report including all the figures and additional details during the summer.

A.1 Master Methods

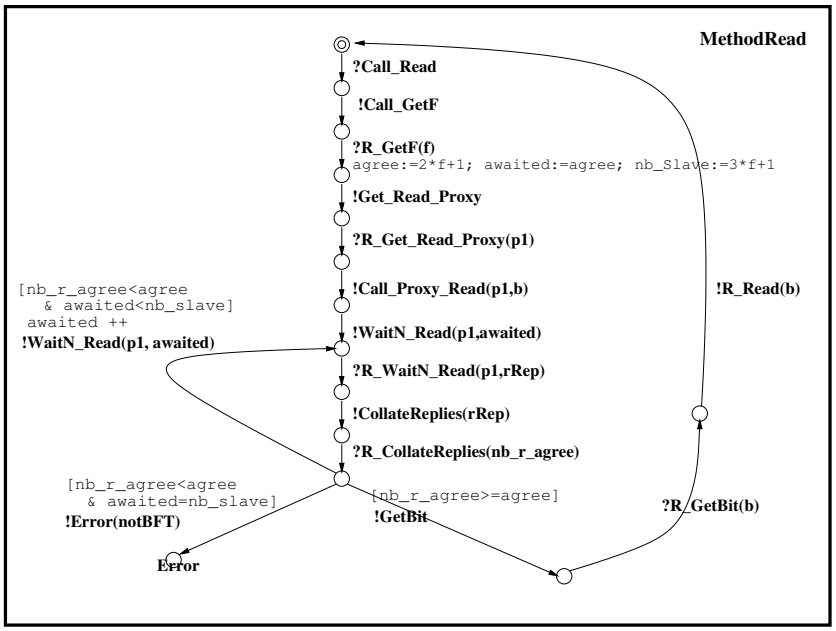

Figure 12: MasterRead Method

The Master Read method. The behaviour of the Read method shown in Fig. 12 is very similar to the Write method in Fig. 6. The main difference is that, after triggering remote invocations and waiting for enough identical replies, it inputs the bit value found by the collate method via the `!GetBit, ?R_GetBit(b)` sequence, the value `b` received is returned to the client.

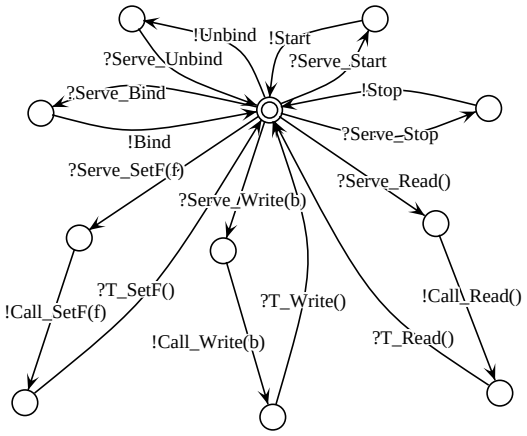

Figure 13: Behaviour of the MasterBody

The master body. The body shown in Fig. 13 is encoded in generic way: it serves sequentially functional and non-functional requests. In this work, we only use the service of each functional request (on method Read, Write). This service calls the adequate method (e.g., `!Call Read`), and waits until the method terminates, signaled by `R` events (e.g., `?R Read`); `R Read` synchronizes both with the component that triggered the request and with the body. As requests are served one after the other, this encodes a mono-threaded behaviour.

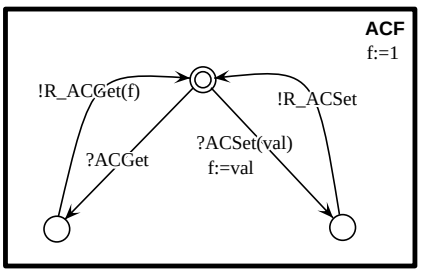

Figure 14: The ACF Process: attribute controller that stores the value of f

The Attribute controller. (see Fig. 14) In Fractal, the attribute controller provides read and write access to the attributes of the components; the only attribute of the Master component is `f`, the number of faults that can be handled. The behaviour of the attribute controller is very simple: it simply provides a setter and a getter method for storing and retrieving the value of `f`. This controller is crucial for the Write and Read methods to know the current value of `f` and to reach a consensus.

A.2 Slave Methods

Body of the slave component. The body of the slave component, Fig. 15, is very similar to the master body, except that the slave only serves requests Read, Commit, and Write.

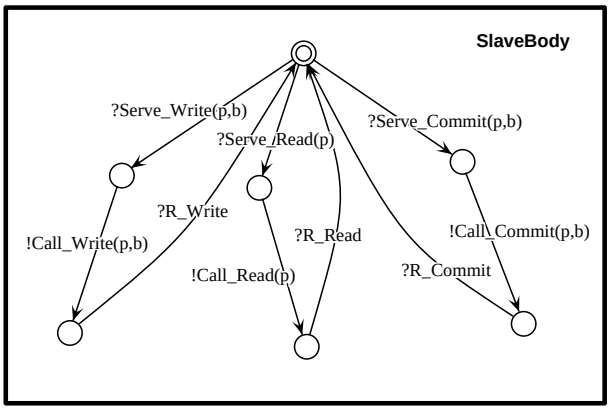

Figure 15: Behaviour for the Body of a slave component

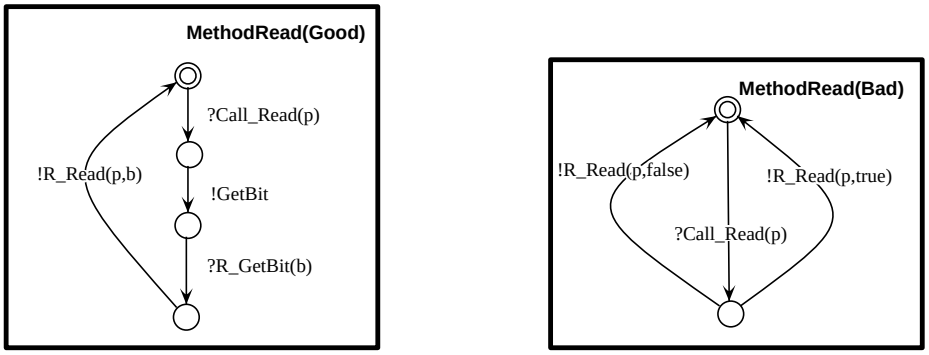

Figure 16: Slave (Good and Bad) Read Method

Read methods of the Slaves The method Read of the good and bad slaves are shown in Figure 16. For the good slave, this method accesses the stored bit by a call to `!GetBit`, and replies with the value that has been read. For a bad slave, the method returns a random value (either `true` or `false`).

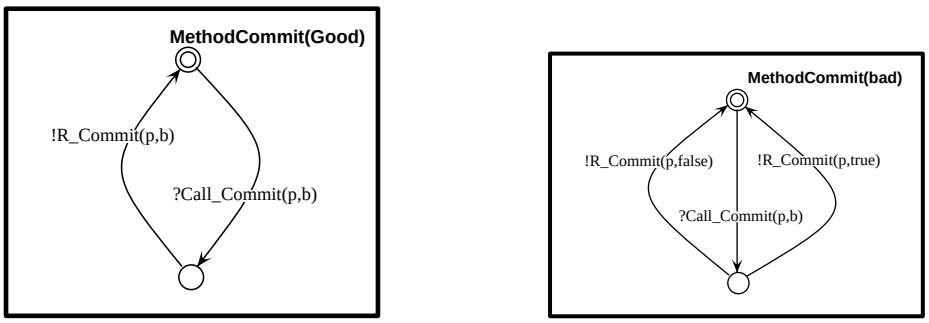

Figure 17: Slave (Good and Bad) Commit Method

Commit methods of the Slaves As stated in the body of the paper the Commit method (Fig.17) has a trivial behaviour and always replies `true` for the good slave,k it replies randomly for the bad one.

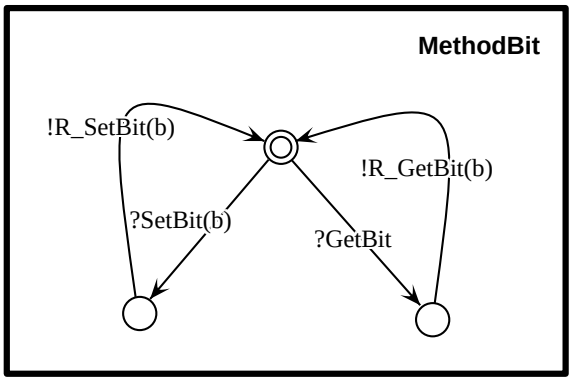

Figure 18: MethodBit process of the good slaves

The Slaves' MethodBit process. Finally the MethodBit process of Fig.18 is used by the good slaves to store and retrieve the value of the stored bit. It provides the two methods `GetBit` and `SetBit`.

B Behavioural Semantics of GCM

We gather here the definition and explanation of the behavioural semantics, in terms of pNets, of various features of GCM systems.

B.1 Graphical versus Formal Presentation of pNets

The pNets model has been formally defined in [1], and a slightly modified (simplified) version in which the notion of *transducer* was removed, in [8]. We use here the [8] definition. One can define a notion of "static" pNet, in which the possible synchronisations do not depend on the state of the transducer (or of specific subnets playing the role of the transducer). Such static pNets are exactely those that can be represented easily by the graphical constructs used in this paper.

The following rules define the mapping between graphics and pNets constructs: formalize... => V2

ASP/ProActive active objects

Fig. 19 illustrates the structure of the pNets expressing a asynchronous communication between 2 active objects. A method call to a remote activity goes through a proxy, that locally creates a "future" object, while the request goes to the remote request queue. The request arguments include the references to the caller and callee objects, but also to the future. Later, the request may eventually be served, and its result value will be sent back and used to update the future value.

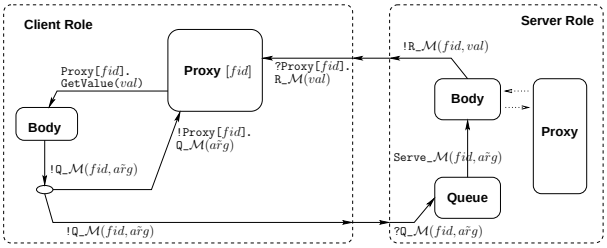

Figure 19: Simple pNets model for the Communication between two Active Objects

Hierarchical Processes

This first construction applies to hierarchical component structures, with interfaces, but none of the specific non-functional features of Fractal or GCM.

From the information in a Component structure, it is straightforward to generate a pNet representing the communication between the interfaces and the subcomponents, from the following elements:

- the pNet has one hole for each (parametric) subcomponent;
- the pNet global actions pA_G and hole sorts $\tilde{O}_J$ are sets of actions of the form $C_i.Itf[!|?]m(a\tilde{r}g)$ for performing / serving a method m with each argument $arg \in \Sigma_{T_{arg},V}$,
- its transducer has one parameterized synchronisation vector for each binding in $\tilde{B}$.

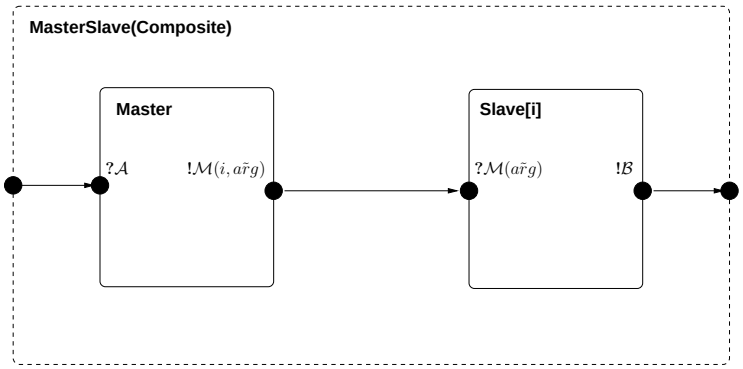

Figure 20: Graphical representation of a parametrized network for a hierarchical component

Graphically a parameterized pNet is represented by a set of boxes (holes). Each box is surrounded by labelled ports encoding a particular Sort (sort constraint pA_G) of the pNet. The box is filled with a pLTS or another pNet (hierarchy) satisfying the Sort inclusion condition ($L \subseteq pA_G$). The ports can be interconnected through edges for synchronization. The edges are labelled with a parametrized action or τ (unobservable action). The edges connecting the boxes express the synchronization between them. Each edge is translated to a synchronization vector of the form $<!\mathcal{M}(a\tilde{r}g), ?\mathcal{M}(a\tilde{r}g) >\rightarrow \mathcal{M}(a\tilde{r}g)$, expressing a rendez-vous between actions $!\mathcal{M}(a\tilde{r}g)$, and $?\mathcal{M}(a\tilde{r}g)$, visible as a global action $\mathcal{M}(a\tilde{r}g)$.

For the example above, the three edges will be translated into the following synchronisation vectors:

$<$Master.$!\mathcal{M}(i, a\tilde{r}g)$, Slave[i].$?\mathcal{M}(a\tilde{r}g) >\rightarrow \mathcal{M}(i, a\tilde{r}g)$

$<$Master.$?\mathcal{A}, - >\rightarrow ?\mathcal{A}$

$< -$,Slave[i].$!\mathcal{B} >\rightarrow !\mathcal{B}(i)$

Structure of GCM Components

We start here with a simple subset of GCM features, excluding any dynamic reconfiguration. The structure of pNets models for such components includes:

- a Request Queue
- a Body encoding the service policy
- for primitive components, pLTSs for each service method s of the component, and eventually for other local methods
- for composite components, children pNets for each subcomponent
- when necessary, Future proxies and Group proxies for each client interface of the component.

The following figures give the overall pNet structure for GCM primitive (resp. composite) component, with edges representing the synchronisation vectors. The pLTS representing the inside of the boxes (Queue, Body, Proxies) will be detailed later.

Primitive component

The pNet for a simple primitive component as depicted in Figure 21 is composed of two parts: the membrane, which will contain the `Queue` and the `Proxy` (corresponding to the $\mathcal{M}2$ method found on the component's client interface), and the content, with the Body and the pLTS for the service method ($\mathcal{M}1$) found on the server interface. The basic communication scenario is as follows: the incoming requests to $\mathcal{M}1$ arrive to the Queue, which will then issue a Serve_$\mathcal{M}1$ request to the Body, which, at its turn, will send the Call request to the box containing the pLTS for the corresponding method ($\mathcal{M}1$), and after the execution of the method has terminated, it will return the result by sending the R_$\mathcal{M}1(val)$ response. The calls to the $\mathcal{M}2$ method will go through the associated `Proxy`, and the `Body` will be able to retrieve the value of the future by calling GetValue($fid2, val$).

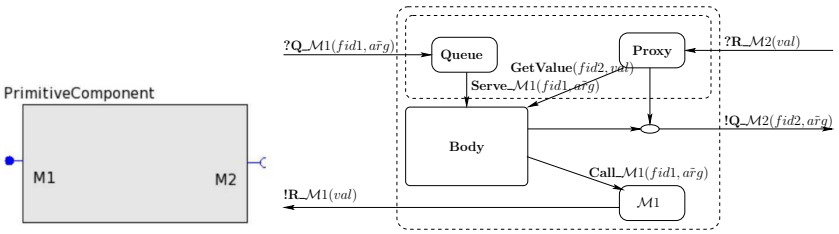

Figure 21: Simple Primitive Component

Composite component

Figure 23 shows the pNets for a composite component shown in Figure 22. The composite component `B` is composed of two subcomponents `C` and `D`, and has two external interfaces (1 client and 1 server). Requests arriving on the server interfaces are of the form `!Q_foo(f)`, the request is enqueued in the queue of `B`, then it will be served (`Serve_foo(f)`) by the body and delegated to `C`. For this a proxy `Proxy(`f_{B^1}`)` waits for the result of the remote call to `C`. Note the synchronisation that occurs at the same time between the proxy `Proxy(`f_{B^1}`)`, the body of the composite and the request queue of `C`. When the request is finished in `C`, the result is returned to the proxy; it can then be obtained by the `Body` (`GetValue_foo(val)`), and returned (`R_foo(f,val)`). The other proxies (another one in `B` and two in `C`) are manipulated similarly. For clarity of the figure, we omitted the communications with proxy `Proxy(`f_{B^2}`)`.

Let us now focus on the remote call from the composite `B` to its subcomponent`C`. Similarly to the composite case, `Q_foo(`f_{B^1}`)` is enqueued in `C`'s queue then served by the body, which delegates two calls, one towards `D`, and the other towards `B` (relying on two different proxies). When the result is obtained it is returned to `Proxy(`f_{B^1}`)` by a communication `R_foo(`f_B`,val)`.

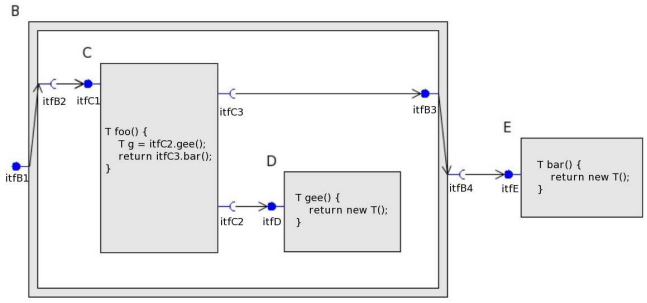

Figure 22: Composite Component

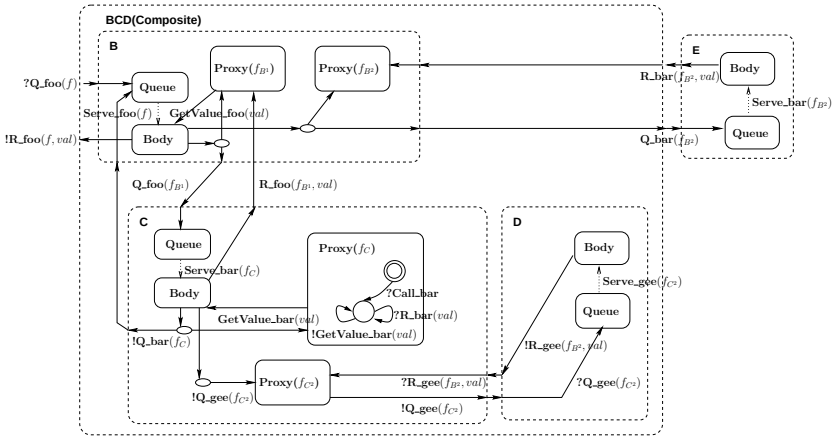

Figure 23: pNet for the Composite Componentof Figure ??, with futures

Body

The body process is the place where the service policy of the component is encoded. By default, this policy picks requests from the Queue in Fifo order (first in first out), and executes them in a monothreaded way. This is represented in Figure 24, where the action `?ServeFirst_`$\mathcal{M}$`(fid,args)` picks the first request from the queue, calls the corresponding service body, then synchronizes with the service termination to send back the result to the caller. Naturally, there will be one such loop for each possible service method in the component.

In the case of a user-defined policy, the application developer will be in charge of providing its specification in the form of a state-machine, that may be "state-full", meaning that the selection of requests in the queue may depend on the internal state of the component. The developer will use the same communication primitives as the defaut Body.

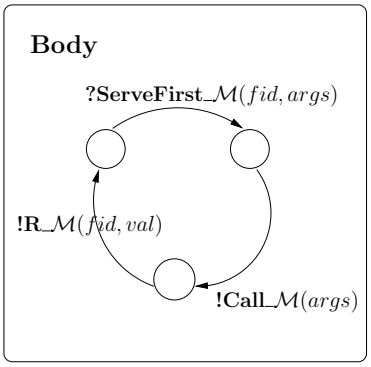

Figure 24: pLTS of the Body

Queue

The Queue pNet encodes an unbounded Fifo queue, containing requests composed by a method name and its arguments, and a selection mode (typically oldest or younguest request matching a predicate).

Typically, we need to instanciate this queue as a bounded structure, before running model-checking tools. However, different model-checkers may use different representation for queues, so we chose to keep an abstract Queue structure in the pNet, and to leave to backend tools the choice of the queue representation.

Future proxies

The behaviour of the future proxy pNet is as follows: the proxy starts by a Call_$\mathcal{M}$ transition. This allows the component to perform the remote call. Then the proxy will wait for the response transition (R_$\mathcal{M}$) to synchronise on the response value. Finally, the component body may access the content of the future through a !GetValue transition.
The second image is an optimized version of the future proxy LTS, that could be used when one proxy would be enough (rather than creating a new proxy each time there is a new future to be handled), so the same one would be recycled and reused several times.

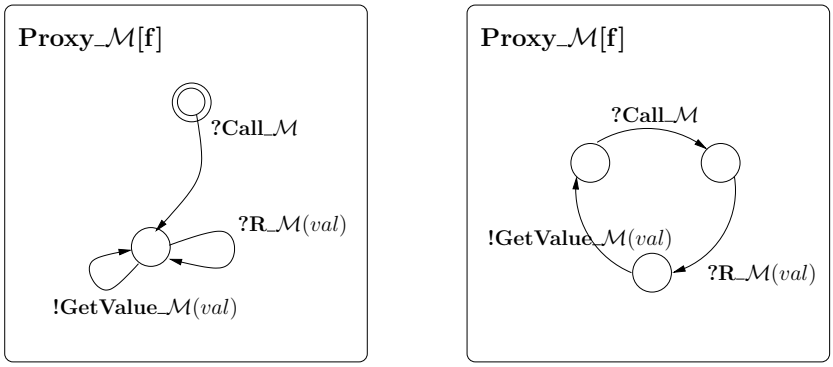

Figure 25: pLTS of the Future Proxy

Group proxies and Proxy manager

Figure 26 is taken from the case-study of this paper, and focuses on the pNet representing the behaviour of a multicast interface. In general, there will be one `Group Manager` for each multicast interface in a component. Each `Group Manager` pNet includes a `Proxy Manager`, which deals with the instantiation/activation of proxies, and one indexed family of proxies for each method in the interface. The `Proxy Manager` answers to requests from the component methods by activating a new proxy for the called method, passing the value of the `current group` to this proxy, and returns the proxy identifier to the calling method. After this, all messages relative to this method call instance (broadcasting the call to the elements of the group, collecting answers, and replying to `Wait` requests from the calling method), are parameterized by this proxy identifier. Whenever the interface receives `Bind` or `Unbind` requests, they modify the "current group" internal variable of the `Proxy Manager` pNet. Remark that this will only affect proxies that will be activated later.

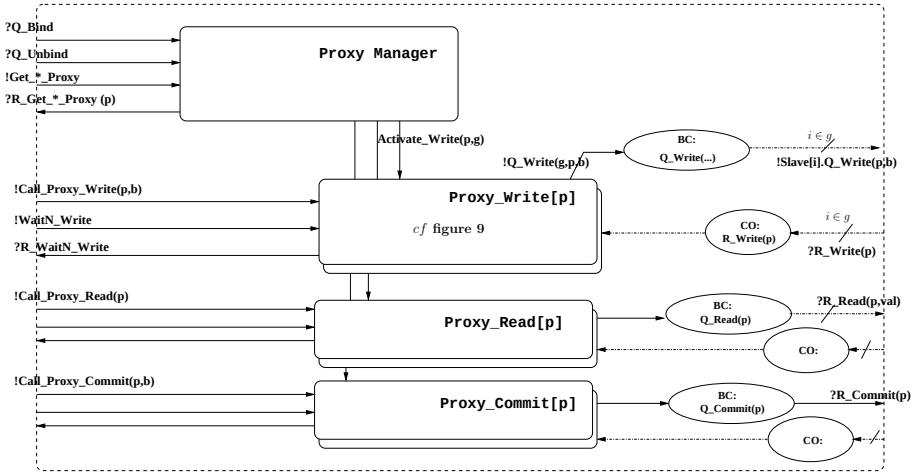

Figure 26: Focus on the elements for managing the group

Attribute Controller

In Fractal, the attribute controller provides read and write access to the attributes of the components; the only attribute of the Master component is `f` – the number of faults that can be handled. The behaviour of the attribute controller is very simple: it simply provides a setter (`ACSet`) and a getter (`ACGet`) method for storing and retrieving the value of `f`.

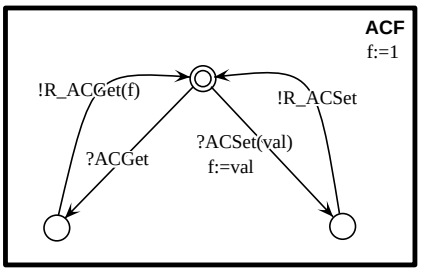

Figure 27: The ACF Process: attribute controller that stores the value of f

Centre de recherche INRIA Sophia Antipolis – Méditerranée
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399