# Behavioural Verification of Distributed Components

Ludovic Henrio and Eric Madelaine
Inria Sophia-Antipolis-I3S-CNRS-University of Nice Sophia-Antipolis
ludovic.henrio@cnrs.fr,eric.madelaine@inria.fr

This paper presents a brief overview of our efforts in the behavioural specification and verification of distributed component systems [1, 2, 4, 5]. Our objective in this work is to provide tools to help the programmer specify the behaviour of his/her components, generate a model, and check the correctness of his/her application.

## 1 Introduction: The Grid Component Model (GCM)

Component models provide a structured programming paradigm, and ensure a very good re-usability of programs. Indeed in component applications, dependencies are defined together with provided functionalities by the means of provided/required ports; this improves the program re-usability. Some component models and their implementations additionally keep a trace at runtime of the component structure and of their dependencies. Knowing how components are composed and being able to modify this composition at runtime provides great adaptation capabilities: the applications can adapt to evolution in the execution environment by changing the components taking part in the composition or their dependencies. GCM [3] has been proposed in the CoreGrid Network of Excellence, it is an extension of the Fractal component model [6] to better address large-scale distributed computing. GCM builds above Fractal and thus inherits from fractal: its hierarchical structure, the enforcement of separation between functional and non-functional concerns, its extensibility, and the separation between interfaces and implementation. The main extensions provided by GCM are:

- GCM supports *collective communications*: one-to-many, and many-to-one.
- GCM also comes with a support for autonomic aspects and better separation between functional and non-functional concerns: more precisely, in GCM non-functional concerns can also be defined as a component assembly.

ProActive/GCM is a reference implementation of the GCM component model that has been implemented during the GridComp European project. It is based on the ProActive Java library and relies on the notion of active objects [9]. It is important to note that each component corresponds at runtime to an active object and consequently each component can easily be deployed on a separate JVM and can be migrated. Of course, this implementation relies on design and implementation choices relatively to the purely structural definition provided by the model. Even if the programming methodology entailed by active objects and GCM is way simpler than RMI-style of programming, bugs are still more frequent in distributed applications than in sequential ones. Indeed, even if our programming model prevent data race-conditions, race-conditions between communications and deadlocks can still exist. The complex interleaving of communications makes the reasoning on a distributed system difficult, even when the system is built from well separated components.

## 2 Approach and contributions

In this work, we focus on the behavioural specification of active object and GCM applications in order to be able to verify their behaviour. The behavioural model we generate is expressed in the pNets formalism[1] that we designed. pNets serves as a low level semantic framework for expressing the behaviour of various classes of distributed languages, and as a common internal

[1] parameterised networks of synchronous automata [2]

To appear in EPTCS.

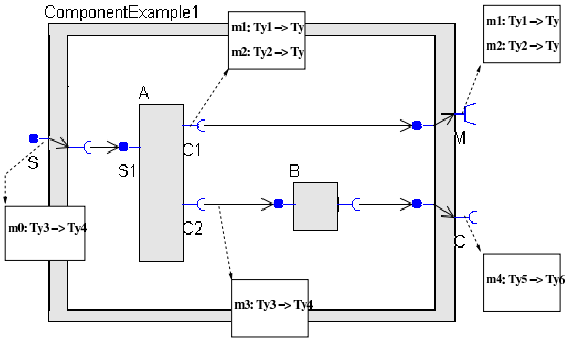


Figure 1: A simple composite component in Vercors

format for our tools. pNets allow the specification of parameterised hierarchical labelled transition systems: classical labelled transition systems can be combined hierarchically, and parameterised by some variables.

Our verification tool is called Vercors.[2] It is a platform that assists the programmer in the specification and the verification of his/her application. It provides tools for specifying an application behaviour: from the behaviour of each service method of the primitive components and a description of the application architecture, the platform is able to generate the behaviour of the whole application. We generate automatically the behaviour for asynchronous communications, queues, futures, and component composition based on the ADL. We have shown also how to extend this approach to the modelisation of first-class futures [3] [8], and of multicast interfaces [5]. Then, from the behavioural model of the application, we are able to verify its properties. In most cases, we chose a finite instantiation domain for the parameters of the pNets, and generated a flat finite labelled transition system on which we can prove the properties of the application by state-of-the art model-checkers. The properties we aim at range from absence of deadlocks, to reachability of some actions, and to "any" temporal property (safety, liveness) specific to the application. Our approach consists in specifying the property to be verified as regular $\mu$-calculus formula [11], or more recently as MCL (Model Checking Language) logics [12] formula. Currently, properties are verified using the CADP toolbox [10], but other verification engines can be considered. In the future, we would like to specify these properties at a higher-level, which would be subject to the same abstractions as the tools we provide to the programmer. First, this includes push-button properties that can be verified automatically on each application like absence of dead-lock. Second, generic properties easy to generate from our tools like reachability of an event should also be supported: the reception of a communication, the emission of a result, ... should also be considered. Also, we should provide support for the expression of more complex properties; they must be expressed with the same abstractions (names, range for parameters, ...) as in the specification tools, which is not yet the case.

Figure 1 shows a composite component drawn in the Vercors platform. The component has two sub-components bound together and to the composite component. Type annotations, attached to each interface, define the signature of each method.

## 3 A behavioural model for active objects

Based on pNets, we first defined a behavioural model for active objects. This model is crucial because it gives a representation, on the form of pNets of most of the features of our programming model:

- *Request queues*, using Queues of pNets, and then instantiated by queues of finite length. In our verified applications, a queue of small length (typically 1 or 2) is sufficient to verify the properties of interest; we also verify that the limit of the queue is never reached to ensure that all the behaviours are explored.
- *A body* serving requests one after the other, and calling an appropriate method.

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

[3] futures are called first-class if they can be passed transparently between distributed entities.

- *Service methods* are supposed to be specified by the programmer, they define the business code of the active object. Of course, an abstract behaviour of the active object should be specified in order to limit the state-space to be explored; it should however be precise enough to allow the verification of all the properties of interest. In particular it should at least contain all the synchronisations between active objects, i.e. request calls and future accesses.
- *Futures* are encoded as proxies, and dedicated communications to return the future values are specified by the synchronisation vectors. In most of our developments, first-class futures were not considered but [8] details our investigations on the treatment of first-class futures in our tools.
  For dealing with futures, we create a family of proxies synchronised first on an initialisation phase upon method call, then with the return of the result at request completion, and finally with the access to the future value upon a wait-by-necessity.

Except for service methods, we defined formally how to generate automatically and generically those pNets. We are currently implementing those pNet generators.

# 4 A behavioural model for GCM

Building models for hierarchical component systems like GCM allows us to adopt a compositional approach, at least from the model specification point of view: we generate behavioural models hierarchically in each component. Even more, if the interface behaviour is further specified, that is to say if the context in which the component will be used is explicitly stated, the behaviour of each component can be generated and reduced, allowing us to envision the exhaustive verification of larger systems.

Below, we give a brief overview of our behavioural models based on our example. Figure 2 shows the pNets structure corresponding to the composite component of Figure 1. It illustrates the structure of the pNets we generate for specifying the behaviour of a composite component. We use it here to illustrate how we are able to generate behavioural models for GCM components. It also shows the synchronisation occurring between different pNets (each synchronisation vector roughly corresponds to one arrow in the drawing). The oval shape is used to represent synchronisations involving more than two processes. This structure can be generated from the structure of the pNet, creating communication channels for requests and

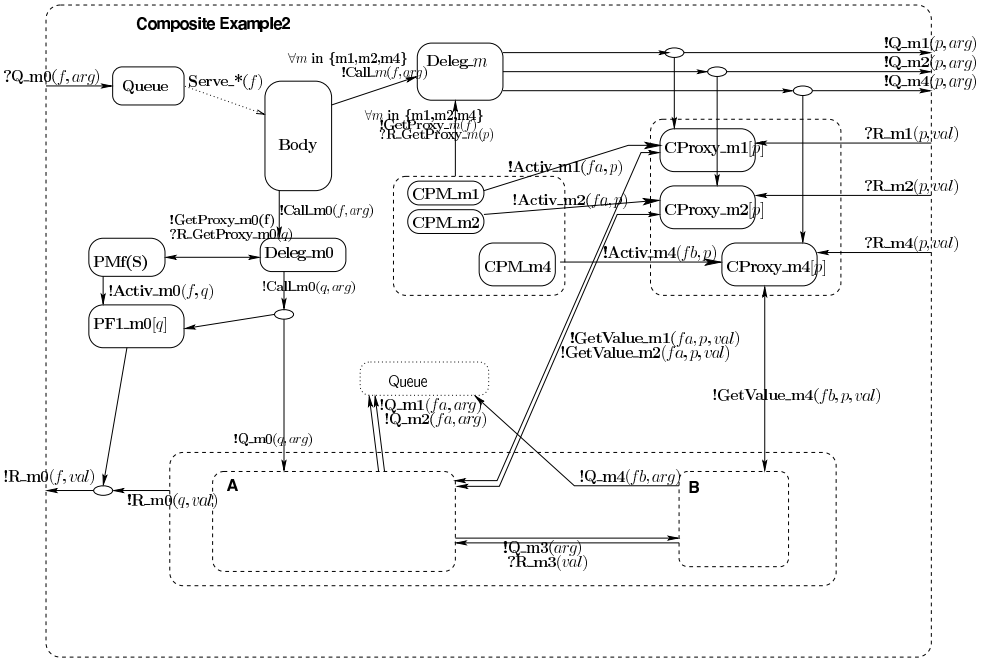


Figure 2: pNet for the composite component from Figure 1

futures for every method provided and required by each component.

Two sub-pNets represent the behaviour of sub-components A and B. A queue pNet receives **?Q_m0(f,arg)** requests where **f** is the future corresponding to the request and **arg** the value passed as argument. **Serve*** communications allow the body to retrieve those requests, which will then be treated by the **Deleg_m0** pNet, this pNet receives **Call** communications from the body and delegates the request to an inner component (here, A); during this process, a future proxy is created by the proxy manager **PMf(S)**, the proxy (**PF1_m0[q]**) is responsible for receiving the reply when **A** has finished the request treatment and for forwarding this result to the outside of the composite component: **R_m0(q,val)** that becomes **R_m0(f,val)**. Note that this proxy encodes some basic form of first class future: the future **q** corresponds to the same result as the future **f**.

Similarly, requests emitted by the inner components arrive in the queue, they are then delegated to the outside world by a similar mechanism: a **Deleg_m** pNet delegates the call, and creates a future proxy, which will be responsible for sending back the result to the appropriate inner component. Here again the proxy manages the fact that both the future **q** and the future **fa** (or **fb**) represent the same result. Finally, note the proxy structure we adopt: there is one proxy manager **CPM*** for each method of each client interface (proxy managers are both indexed over interfaces and over methods). Then each of those managers itself manages a family of proxies **CProxy***. Performing model-checking on (the behaviour of) those structures then requires a precise definition and optimisation of the number and size of those families.

All the communications expressed above, but also the communication channels between the different inner components – requests **Q_m3** and the corresponding replies **R_m3** – can be automatically generated and correspond to synchronisation vectors of the pNet of the composite. Different boxes are expressed as pLTSs[4], except of course inner components that are pNets. Those pLTSs are not shown here for conciseness of this section. Similarly, we are able to generate communication specific aspects: queues, bodies, and future proxies for GCM components.

To illustrate the kind of properties we are able to verify, we proved by model-checking that a fault-tolerant application consisting of 1 master and 4 slaves [5] behaves correctly: 1) it answers to requests: we proved both reachability and (fair) inevitability of termination of services, 2) the answers (values returned by services) are correct. We can also check that the assumptions we make are verified in practice. Here we have proved that a queue depth of 1 is sufficient to prove all of our correctness properties: `< true* . 'Error (Master-OutOfBounds)'> true`

## 5 Related work

The closest work to ours are the frameworks dedicated to the verification of behavioural properties on component applications, we focus below on a detailed comparison with two approaches: SOFA and Symbolic Transition Systems. A more exhaustive study of related works and positioning can be found in [2].

The SOFA system [7] is a development and verification framework for large-scale distributed software systems based on hierarchical components. It uses *behaviour protocols* [13] to specify interactions between components in terms of ordering of method invocation events. The behaviour compliance and consent relations are defined on behaviour protocols based on their trace semantics, allowing reasoning on substitutability and compositional compatibility. The *frame protocol* defines the behaviour of a component. In a composite component, the behaviour is constructed from frame protocols of its subcomponents, and checked for compliance with the composite frame protocol. For a primitive component, the Java implementation may be checked for compliance with a model checking tool.

Symbolic Transition Systems (STS) [14], are structures akin to our pNets. In the STSLib toolset, there is a dedicated specification language (with abstract data types) for distributed components, that are modelled by STS, themselves mapped to LOTOS programs that can be model-checked with the CADP verification toolset [10]. STS do not use the distinction between required and provided ports (or interfaces), whereas it is one of the main building blocks of our component systems. In fact, communication is not based on the classic notion of method calls, but on messages in which both parties (emitter and receiver) must agree in order to communicate. Although this adds expressivity to the language, it also has an impact on the asynchrony of

[4] a pLTS is a LTS with parameters and variables

the system. In our approach, we write only synchronisation vectors corresponding to the semantics of the ProActive library. Our specification language is more independent from the middleware, it allows us to express complex synchronisations that cannot happen in ProActive. This allows us to reason on efficient, expressive, and proved communication mechanisms. Overall, even if pNets formalism is approximately at the same level of abstraction as STS, in our approach, the programmer is rather exposed to a higher-level composition framework, closer to his/her usual programming and composition concerns.

## 6 Conclusion

We presented an overview of the behavioural verification efforts we have been doing in the OASIS team. We now are able to give a representation of a component system behaviour including its structural aspects, the handling of asynchronous requests and of futures, the hierarchical composition of components, and group communications. Overall, from the specification of service methods and the description of the component architecture, we are able to generate a pNet specifying the behaviour of a component assembly. Then, from finite instantiation domains for future proxies, queue length, ... we are able to generate a finite behavioural model that can be model-checked to verify the correct behaviour of a GCM application. More details on the size of the systems we are able to verify and the optimisation techniques we rely on can be found in [4, 5].

## References


[1] Rabéa Ameur-Boulifa, Ludovic Henrio, Eric Madelaine & Alexandra Savu (2012): *Behavioural Semantics for Asynchronous Components*. Rapport de recherche RR-8167, INRIA.

[2] Tomás Barros, Rabéa Boulifa, Antonio Cansado, Ludovic Henrio & Eric Madelaine (2009): *Behavioural Models for Distributed Fractal Components*. *Annals of Telecommunications* 64(1–2). Also Research Report INRIA RR-6491.

[3] Françoise Baude, Denis Caromel, Cédric Dalmasso, Marco Danelutto, Vladimir Getov, Ludovic Henrio & Christian Pérez (2009): *GCM: A Grid Extension to Fractal for Autonomous Distributed Components*. *Annals of Telecommunications* 64(1), pp. 5–24.

[4] Rabéa Boulifa, Ludovic Henrio & Eric Madelaine (2010): *Behavioural Models for Group Communications*. In: *WCSI-10: International Workshop on Component and Service Interoperability*, *EPTCS* 37, pp. 42–56.

[5] Rabéa Ameur Boulifa, Raluca Halalai, Ludovic Henrio & Eric Madelaine (2011): *Verifying Safety of Fault-Tolerant Distributed Components*. In: *International Symposium on Formal Aspects of Component Software (FACS 2011)*, Lecture Notes in Computer Science, Springer, Oslo.

[6] Eric Bruneton, Thierry Coupaye, M. Leclercp, V. Quema & Jean Bernard Stefani (2004): *An Open Component Model and Its Support in Java*. In: *7th Int. Symp. on Component-Based Software Engineering (CBSE-7)*, LNCS 3054.

[7] T. Bures, P. Hnetynka & F. Plasil (2006): *Sofa 2.0: Balancing advanced features in a hierarchical component model*. In: *Software Engineering Research, Management and Applications, 2006. Fourth International Conference on*, IEEE, pp. 40–48.

[8] Antonio Cansado, Ludovic Henrio & Eric Madelaine (2008): *Transparent First-class Futures and Distributed Component*. In: *International Workshop on Formal Aspects of Component Software (FACS'08)*, Electronic Notes in Theoretical Computer Science (ENTCS).

[9] Denis Caromel, Ludovic Henrio & Bernard Paul Serpette (2008): *Asynchronous sequential processes*. *Information and Computation* Volume 207, Issue 4.

[10] H. Garavel, F. Lang & R. Mateescu (2002): *An Overview of CADP 2001*. *European Association for Software Science and Technology (EASST) Newsletter* 4, pp. 13–24.

[11] D. Kozen (1985): *Results on the Propositional Mu-Calculus*. *Theoretical Computer Science* 40.

[12] R. Mateescu & D. Thivolle (2008): *A Model Checking Language for Concurrent Value-Passing Systems*. In K. Sere J. Cuellar, T. S. E. Maibaum, editor: *FM'08*, *LNCS* 5014, Springer, Heidelberg.

[13] F. Plasil & S. Visnovsky (2002): *Behavior Protocols for Software Components*. *IEEE Transactions on Software Engineering* 28(11).

[14] P. Poizat, J.C. Royer & G. Salaun (2006): *Bounded Analysis and Decomposition for Behavioural Descriptions of Components*. In: *FMOODS*, *LNCS 4037*.