Specification and Verification for Grid Component-Based Applications: From Models to Tools

Antonio Cansado and Eric Madelaine

INRIA – CNRS – I3S – Université de Nice Sophia-Antipolis
2004 Route des Lucioles, Sophia Antipolis - France
{acansado,madelain}@sophia.inria.fr

Abstract. Computer Grids offer large-scale infrastructures for computer intensive applications, as well as for new service-oriented paradigms. Programming such applications brings a number of difficulties due to asynchrony and dynamicity, and require specific verification methods. We define a behavioural model called pNets for describing the semantics of distributed component systems. pNets (for parameterized networks of synchronised automatas) are hierarchical assemblies of labelled transition systems, with data parameters expressing both value-passing and parameterized topology. We use pNets for building models for Fractal (hierarchical) and GCM (distributed) components. We present the VerCors platform, that implements these model generation procedures, but also abstraction mechanisms and connections with the model-checking engines of the CADP toolset.

1 Introduction

Software components [1] are the de facto standard in many information technology industries. Component-based frameworks and languages are seen as the natural successors of object-oriented languages for obtaining applications which are more modular, composable and reusable. Many solutions have been proposed during the past 10 years, with EJB being certainly the most well-known and used one. However, these promises are often considered from a software engineering perspective and are at best only empirically verified. We want to build development methods and environments that allow application designers to specify the external behaviour of software components in a black-box fashion, assemble them to build bigger components while guaranteeing that the parts will behave smoothly together, and check that such an assembly implements the overall behaviour expected by the user requirements. Beyond interoperability between components constituting large modern systems, e.g. in grid computing applications, or in large scale distributed software services, raise additional problems. In particular distributed and asynchronous components require more complex behaviour models, and the complexity of the analysis is higher. The analysis of properties related with reconfiguration and dynamicity brings new aspects to check, e.g. defining evolving systems, or checking substitutability.

Among the existing component models, *Fractal* [2] provides the following crucial features: the explicit definition of provided/required interfaces for expressing dependencies between components; a hierarchical structure allowing to build components

F.S. de Boer et al. (Eds.): FMCO 2008, LNCS 5751, pp. 180–203, 2009.

by composition of smaller components; and the definition of non-functional features through specific interfaces, providing a clear separation of concerns between functional and non-functional aspects. The *Grid Component Model (GCM)* [3], extends Fractal by addressing large scale distributed aspects of components, providing structures for asynchronous method calls with implicit futures[1], and NxM communication mechanisms. Both Fractal and GCM models provide means to specify and implement management and reconfiguration operations.

The objective of our work is to provide tools to the programmer of distributed components systems in order to verify the correct behaviour of programs. We require those tools to be intuitive and user-friendly to be usable by non-experts of formal methods. To this end we build an analysis toolset, including graphical editors for defining the architecture and the behaviour of components, and state-of-the-art model-checking tools. At the heart of this platform lie the behaviour semantics of our component systems, and the model generation tools that are the subject of this article. In this context the choice of the behavioural model is crucial: it has to be compact, expressive enough to represent the behavioural semantics, but not too much, that could prevent us to map the models to the input formats of automatic verification tools. Some recent approches, for example π-ADL [4], are using formalisms based on the π-calculus, others, like μ-CRL [5] or STS [6] use algebraic descriptions of data domains. In both cases, such foundations give them powerful primitives for describing dynamic or mobile architectures, but also strong limitations for using automatic verification.

Most established approaches, on the other side, are using intermediate formats with data, that can be unfolded to finite-state structures. This is the case e.g. for the CADP toolbox [7], or for the SPIN model-checker and its specification language PROMELA, whose data values are instantiated (on bound domains) by the state exploration engines.

Our choice is to use an intermediate approach with a compositional semantic model including data called pNets [8]. It is different from previous approaches in the sense that we want a low-level model able to express various mechanisms for distributed systems, and that we do not limit ourselves to finite systems: we shall be able to define mappings to various classes of systems, finite or not. At the same time, the structure of our parameterized model is closer to the programming language or the specification language structure. Consequently, parameterized models are more compact, and easier to produce, than classical internal models. Typically, our pNets model is lower level than Lotos and Promela, but more flexible for expressing different synchronisation mechanisms. On the other hand, it has no recursive constructs, in order to better control the finiteness of encodings.

The second half of this work is a set of software tools called VerCors [9] for specifying and verifying GCM component systems. In the middle term, it will include both a textual and a graphical specification languages, unifying the architectural and the behavioural description of components [10]. It provides tools for defining abstractions of the system, and for computing their behaviour model in term of pNets. Finally it

[1] This is in contrast with languages like MultiLisp or Creol, where futures are explicit in the code. Having implicit futures in GCM/ProActive allows us to automatically provide optimal asynchrony.

has bridges with the CADP verification toolset, allowing efficient (explicit) state-space construction, and model-checking.

In the next section we describe the context of this work, namely the formalisms and models that we use for hierarchical distributed components: Fractal and GCM, and the communication mechanisms of the GCM implementation ProActive. In section 3 we recall the definitions of the parameterized networks of synchronised automatas (pNets), and we give the definition of the behavioural semantics of distributed components, starting with active objects, then modelling hierarchical components, Fractal components, and finishing with the specific features of GCM components, including multicast and gathercast interfaces, and first-class futures. In section 4, we describe the VerCors specification and verification platform, with a glimpse at its architecture, a description of the graphical editors, of the model generation tool, and some results obtained with the platform.

2 Context: Asynchronous Component Model, Active Objects, Grids

2.1 ASP and Active Objects

The ASP calculus [11] is a distributed object calculus with futures featuring:

- asynchronous communications: by a request-reply mechanism,
- futures, that are promised replies of remote method invocations,
- sequential execution within each process: each object is manipulated by a single thread of control,
- imperative objects: each object has a state.

An essential design decision is the absence of sharing: objects live in disjoint activities. An activity is a set of objects managed by a unique process and a unique active object. Active objects are accessible through global/distant references. They communicate through asynchronous method calls with futures. A future is a global reference representing a result not yet computed. The main result consists in a confluence property and its application to the identification of a set of programs behaving deterministically. This property can be summarized as follows: future updates can occur at any time; execution is only characterized by the order of requests; programs communicating over trees are deterministic.

From the proposed framework, we have shown a path that can lead to a component calculus [12]. It demonstrates how we can go from asynchronous distributed objects to asynchronous distributed components, including collective remote method invocations (group communications), while retaining determinism.

The impact of this work on the development of the ProActive library on one hand, and on the building of the behavioural semantics on the other hand, is probably one of our strongest achievements.

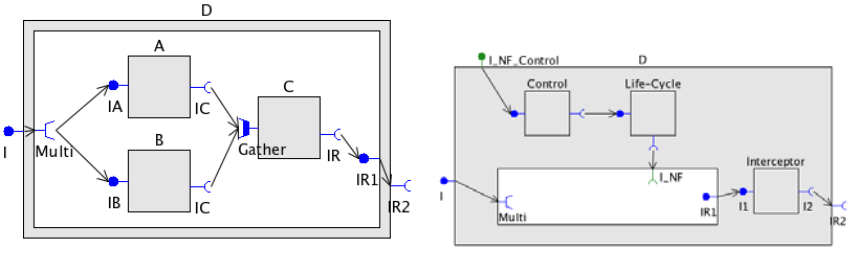
2.2 Fractal and GCM

Fractal [2] is a flexible and extensible component model. Its main features are: a hierarchical structure, in which everything can be built from components (including bindings and membranes), a generic description of non-functional concerns (e.g. life-cycle,

binding, attribute management) through specific control interfaces, a strong separation of concerns between functional and non-functional aspects, a well-defined architecture description language (ADL), and several implementations [13, 14].

The Grid Component Model (GCM) [3] is a novel component model that has been defined by the European Network of Excellence CoreGrid and implemented by the EU project GridCOMP. The GCM is based on Fractal, and extends it to address Grid concerns.

Grids consider thousands of computers all over the world; programming Grids involve dealing with latency in communications between computing nodes, and optimizing whenever possible the parallelism of the computation. For that, GCM extends Fractal using asynchronous method calls. Grid applications usually have numerous similar components, so the GCM defines collective interfaces which ease design and implementation of such parallel components by providing synchronisation and distribution capacities. There are two kinds of collective interfaces in the GCM: multicast (client) and gathercast (server).



(a) Content of a composite component (b) Membrane of a component

Fig. 1. GCM components

One to N and N to one interfaces. Typically a multicast interface (such as the interface `Multi` in Fig. 1(a)) is bound to the service interfaces of a number of parallel components, and a method call toward this interface is distributed, as well as its parameters, to several or all of them. GCM provides various policies for the request parameters, that can be broadcast, or scattered, or distributed in a round-robin fashion; additional policies can be specified by the user. The computation on the remote components will eventually terminate and send back, asynchronously, their results; Then the results of the invocations have to be assembled back with different possible policies (gather the results in a list, return the sum of the results, compute the maximum, or just pick the first that arrives and discard others...).

Symmetrically, gathercast interfaces (e.g. `Gather` in Figure 1(a)) are bound to a number of client components, and various synchronisation policies are provided. This corresponds to synchronisation barriers in message-based parallel programming, though here you may also have to specify how you redistribute the result on the client interfaces.

This treatment of collective communications provides a clear separation of concern between the programming of each component, and the management of the application topology: within a component code, method calls are addressed simply to the component local interfaces. The management of bindings of clients (on a gathercast interface) or services (on a multicast interface) is separated from the functional code.

Membranes and Non-functional interfaces. The component's non-functional (NF) aspects are handled by the component's membrane. The membrane is structured as a component system defining so-called *NF components*. Moreover, the GCM specifies interfaces for the autonomic management and adaptation of components. The membrane is also in charge of controlling the interaction between the component's content and the environment: the membrane decides how requests entering or leaving the component are to be treated.

The simplest binding one can define in a membrane is a binding from an external interface to an internal interface (e.g server interface I to internal interface `Multi` in Figure 1(b)): requests will simply be forwarded to a subcomponent server interface. But a NF component called `Interceptor` can be inserted between an external and an internal functional interface that will perform some non-functional processing (e.g. encrypting, logging, etc); an example is the `Interceptor` component between interfaces `IR1` and `IR2` in Fig. 1(b)).

More complex NF components can be used for introspection, reconfiguration, or autonomic management. Those will typically lie between the external and internal NF interfaces of the composite component.

Architecture. The Architecture Description Language (ADL) of both Fractal and the GCM is an XML-based format, that contains both the structural definition of the system components (subcomponents, interfaces and bindings), and some deployment concerns. Deployment relies on *virtual nodes* that are an abstraction of the physical infrastructure on which the application will be deployed. The ADL only refers to an abstract architecture, and the mapping between the abstract architecture and a real one is given separately as a deployment descriptor.

The Fractal/GCM ADL descriptions are static. Dynamicity of component applications, and the ability to reconfigure them, is gained through specific operations of their APIs. Several aspects of GCM, including its ADL, API, deployment description, application resources description, are now standardized by the European Telecommunication Standards Institute ETSI [15].

2.3 A GCM Reference Implementation: GCM/ProActive

The GCM reference implementation is based on ProActive [16], an Open Source middleware implementing the ASP calculus. In this implementation, an active object is used to implement each primitive component and each composite membrane. Although composite components do not have functional code themselves, they have a membrane that encapsulates controllers, and dispatches functional calls to inner subcomponents. As a consequence, this implementation also inherits some constraints and properties w.r.t. the programming model:

- components communicate through asynchronous method calls with transparent futures (place-holders for promised replies): a method call on a server interface adds a request in the server's *request queue*;
- communication semantics use a "rendez-vous" ensuring the causal ordering of communications;

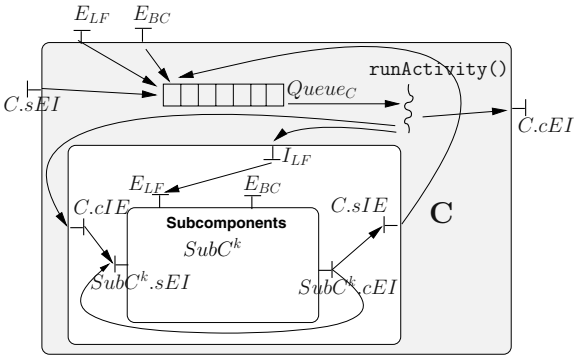


Fig. 2. *ProActive* composite component

– synchronisation between components is ensured with a data-flow synchronisation called *wait-by-necessity*: futures are first order objects that can be forwarded to any component in a non-blocking manner, execution is only blocked if the concrete value of the result is needed (accessed), while the result is still unavailable;
– there is no shared memory between components, and a single thread is available for each component.

Each primitive component is associated with an active object written by the programmer. Some methods of this active object are exported as the methods of the component's interfaces. The active object managing a composite is generic and provided by the GCM/ProActive platform; it forwards the functional requests it receives to its subcomponents. Primitive component functionalities are addressed by the encapsulated active object. For primitive components, it is possible to define the order in which requests are served by writing a specific method called `runActivity()`; we call this the service policy. If no `runActivity()` is given, a default one implements a FIFO policy (excepted for non-functional requests, see below). Composite components always use a FIFO policy. Note that futures create some kinds of implicit return channels, which are only used to return one value to a component that might need it.

Life-Cycle of GCM/ProActive Components. GCM/ProActive implements the membrane of a composite as an active object, thus it contains a unique request queue and a single service thread. The requests to its external server interfaces (including control requests) and from its internal client interfaces are dropped to its request queue. A graphical view of a composite is shown in Fig. 2.

Like in Fractal, when a component is stopped, only control requests are served. A component is started by invoking the non-functional request: `start()`. Because threads are non-interruptible in Java, a component necessarily finishes the request it is treating before being stopped. If a `runActivity()` method is specified by the programmer, the stop signal must be taken into account in this method.

Note that a *stopped* component will not emit functional calls on its required interfaces, even if its subcomponents are active and send requests to its internal interfaces.

2.4 Example

We will use the example in Fig. 3 to illustrate the various aspects of this paper. It is formed from one composite component B and three primitive components A, C, D. Component B has a number of subcomponents, and requests on its server interface S are dispatched to them through the multicast interface MC. Component D has two server interfaces W and R, and is supposed to host some shared resource (e.g. a database); its role in the example is to show the possible race-conditions or deadlocks that could arise, e.g, if a request on interface W has a side effect on the shared resource. Component A plays the client role, and will send requests to B, creating futures containing their promised responses, and transmitting these futures as parameters to requests to C. Component B also has two non-functional interfaces NF1 and NF2 that may be used e.g. to reconfigure its content.

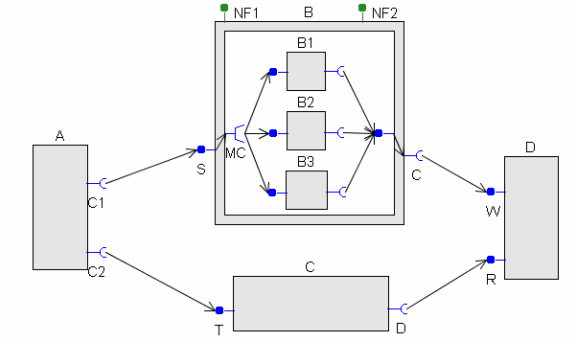


Fig. 3. Running example

3 Semantic Model

In this section, we recall the main definitions of the *parameterized Networks of synchronised automatas* (pNets, [8]). We use pNets as a general low level behaviour model for encoding different variants of our languages or component models. We start with the formal definitions of the model. Then we use pNets to define the behavioural semantics of two basic and important formalisms in the domain of distributed components: the ProActive "Active Objects" on one hand, and Fractal hierarchical components on the other hand (both examples are excerpts from [8]). Finally, we give an encoding for GCM components, including the management of request queues in primitives and composite components, and the encoding of future proxies, in presence of first class futures.

3.1 Parameterized Networks of Synchronised Automata (pNets)

The following definitions are taken from [8]. We start with classical labelled transition systems and structure them using synchronisation networks. Then we extend these definitions to include parameters, both as arguments in communication and in state definitions (à la "value-passing CCS"), and in synchronisation operators, obtaining a model powerful enough to describe parameterized and dynamic topologies.

We model the behaviour of a process as a Labelled Transition System (**LTS**) in a classical way [17]. The LTS transitions encode the actions that a process can perform in a given state.

Definition 1. LTS. *A labelled transition system is a tuple* $\langle S, s_0, L, \rightarrow \rangle$ *where* S *(possibly infinite) is the set of states,* $s_0 \in S$ *is the initial state,* L *is the set of labels,* $\rightarrow$ *is the set of transitions :* $\rightarrow \subseteq S\,xLxS$. *We write* $s \xrightarrow{\alpha} s'$ *for* $(s, \alpha, s') \in \rightarrow$.

We define **Nets** in a form inspired by the *synchronisation vectors* of Arnold and Nivat [18], that we use to synchronise a (potentially infinite) number of processes.

In the following definitions, we frequently use indexed vectors: we note $\tilde{x}_I$ the vector $\langle ..., x_i, ... \rangle$ with $i \in I$, where I is a countable set.

Definition 2. Network of LTSs.[2] *Let Act be an action set. A* **Net** *is a tuple* $\langle A_G, J, \tilde{O}_J, \overrightarrow{V} \rangle$ *where* $A_G \subseteq Act$ *is a set of global actions,* J *is a countable set of argument indexes, each index* $j \in J$ *is called a* hole *and is associated with a* sort $O_j \subset Act$. $\overrightarrow{V} = \{\overrightarrow{v}\}$ *is a set of synchronisation vectors of the form:* $\overrightarrow{v} = \langle a_g.\tilde{\alpha}_I \rangle$ *where* $a_g \in A_G$, $I \subseteq J \wedge \forall i \in I, \alpha_i \in O_i$

Fig. 4 gives a naive representation of the Net representing component B, with four subcomponents. Here the semantics has been configured so that `call` requests are going through a MC policy component, and are made visible (to the next level) as "`?call(m,args)`" for requests received by `B`, and "`B[i].call(m,args)`" for the requests dispatched to the respective `B[i]`. As an example, the second synchronisation vector in $\overrightarrow{V}$ reads as: action "`!call(m,x1)`" of the first hole (here `MC`) can occur synchronised with action "`?call(m,x1)`" of `B1`, and the corresponding global action is "`B[1].call(m,x1)`". There should be one such vector for each possible value of `x1`.

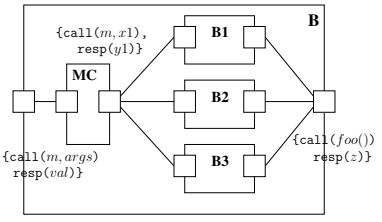
Note that the specific syntax (and meaning) of the actions is not important here: it depends on the specific formalism that has been translated into Nets. The synchronisation vectors are the only means that we use to express the synchronisation mechanisms. This way we can express traditional message passing (matching emission/reception), as well as other mechanisms like one to N synchronisation. In this first non parameterized version, we may need a infinite number of vectors to express the synchronisations occuring in a Net.

Definition 3. *A* **System** *is a tree-like structure whose nodes are* **Nets***, and leaves are* **LTSs***. At each node a partial function maps holes to corresponding* **subsystems***. A system is* **closed** *if all holes are mapped, and* **open** *otherwise.*

Definition 4. *The* **Sort** *of a system is the set of actions that can be observed from outside the system. It is determined by its top-level node, with:*

$$Sort(\langle S, s_0, L, \rightarrow \rangle) = L \qquad Sort(\langle A_G, J, \tilde{O}_J, \overrightarrow{V} \rangle) = A_G$$

[2] This definition is simpler than the one we gave in [8], from which we have removed the *transducer* element in the pNet structure. It is possible to obtain an expressiveness similar to pNets with transducers by adding an extra argument to each pNet, and specifying this "Controller" as an argument pLTS.



where *B-3-Net* = $\langle A_G, J, \tilde{O}_J, \overrightarrow{V} \rangle$ with:

A_G = {`?call(m,args), !resp(val), B1.call(m,x), ...`}

J = {`MC, B1, B2, B3`}

O_{MC} = {`?call(m,args), !resp(val), !call(m,x1), ...`}

$O_{B1} = O_{B2} = O_{B3}$ = {`?call(m,x), !resp(val), !call(foo()), ?resp(z)`}

$\overrightarrow{V}$ ={

```
⟨ ?call(m,args), ?MC.call(m,args), -, -, -⟩
⟨ B[1].call(m,x1), !B1.call(m,x1), ?call(m,x1), -, -⟩
⟨ B[2].call(m,x2), !B2.call(m,x2), -, ?call(m,x2), -⟩
... }
```

Fig. 4. Example of Net

Next we enrich the above definitions with parameters in the spirit of Symbolic Transition Graphs [19]. We start by giving the notion of parameterized actions. We leave unspecified here the constructors and operators of the action algebra, they will be defined together with the encoding of some specific formalism.

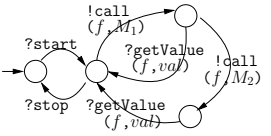
Definition 5. Parameterized Actions. *Let P be a set of names, $\mathcal{L}_{A,P}$ a term algebra built over P, including at least a distinguished sort Action, and a constant action τ. We call $v \in P$ a parameter, and $a \in \mathcal{L}_{A,P}$ a parameterized action, $\mathcal{B}_{A,P}$ the set of boolean expressions (guards) over $\mathcal{L}_{A,P}$.*

Definition 6. pLTS. *A parameterized LTS is a tuple $\langle P, S, s_0, L, \rightarrow \rangle$ where:*

- *P is a finite set of parameters, from which we construct the term algebra $\mathcal{L}_{A,P}$,*
- *S is a set of states; each state $s \in S$ is associated to a finite indexed set of free variables $fv(s) = \tilde{x}_{J_s} \subseteq P$,*
- *$s_0 \in S$ is the initial state,*
- *L is the set of labels, $\rightarrow$ the transition relation $\rightarrow \subset S \times L \times S$*
- *Labels have the form $l = \langle \alpha,\ e_b,\ \tilde{x}_{J_{s'}} := \tilde{e}_{J_{s'}} \rangle$ such that if $s \xrightarrow{l} s'$, then:*
  - *α is a parameterized action, expressing a combination of inputs $iv(\alpha) \subseteq P$ (defining new variables) and outputs $oe(\alpha)$ (using action expressions),*
  - *$e_b \in \mathcal{B}_{A,P}$ is the* optional *guard,*
  - *the variables $\tilde{x}_{J_{s'}}$ are assigned during the transition by the* optional *expressions $\tilde{e}_{J_{s'}}$*

  with the constraints: $fv(oe(\alpha)) \subseteq iv(\alpha) \cup \tilde{x}_{J_s}$ and $fv(e_b) \cup fv(\tilde{e}_{J_{s'}}) \subseteq iv(\alpha) \cup \tilde{x}_{J_s} \cup \tilde{x}_{J_{s'}}$.

Example: Fig. 5 represents a possible behaviour of the body of component A from our example. The action alphabet used here reflects the active object communication schema: each remote request sent by the body has the form "`!call(`f, $\mathcal{M}(\tilde{arg})$`)`", where $\mathcal{M}$ is the method name, eventually with parameters $\tilde{arg}$, and f is the identifier of the future proxy instance. Thus in this example, the action expressions are built from variables f and *val*, from the constants M_1 and M_2, and from the binary action constructors `call` and `getValue`. These actions allow the component to perform a remote method call, and



$A\text{-}LTS = \langle P, S, s_0, L, \rightarrow \rangle$
with:

$P = \{f, val\}$

$S = \{s_i\}$, `i ∈ [0:3]`

$L = \{$ `?start,?stop,!call(`f`,`M_1`),!call(`f`,`M_2`),?getValue(`f`,`val`)` $\}$

$\rightarrow$ such that:

s_0 : `?start` $\rightarrow s_1$,
s_1 : `?stop` $\rightarrow s_0$,
s_1 : `!call(`f`,`$M1$`)` $\rightarrow s_2$,
s_2 : `?getValue(`f`,`val`)` $\rightarrow s_1$
s_3 : `!call(`f`,`$M2$`)` $\rightarrow s_3$
s_4 : `?getValue(`f`,`val`)` $\rightarrow s_1$

Fig. 5. Behavioural model of component A

access the return value resp.; more details on how the component communicates with its environment are given later in Fig. 7.

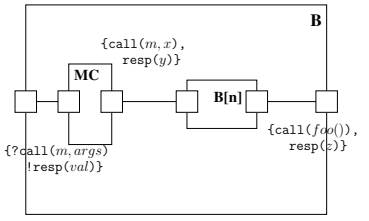
Now, we define pNets as Nets where the holes can be indexed by a parameter, to represent (potentially unbounded) families of similar arguments.

Definition 7. *A* **pNet** *is a tuple* $\langle P, pA_G, J, \tilde{p}_J, \tilde{O}_J, \overrightarrow{V} \rangle$ *where: P is a set of parameters,* $pA_G \subset \mathcal{L}_{A,P}$ *is its set of (parameterized) external actions, J is a finite set of holes, each hole j being associated with (at most) a parameter* $p_j \in P$ *and with a sort* $O_j \subset \mathcal{L}_{A,P}$. $\overrightarrow{V} = \{\overrightarrow{v}\}$ *is a set of synchronisation vectors of the form:* $\overrightarrow{v} = \langle a_g, \{\alpha_t\}_{i \in I, t \in B_i} \rangle$ *such that:* $I \subseteq J \wedge B_i \subseteq \mathcal{D}om(p_i) \wedge \alpha_i \in O_i \wedge fv(\alpha_i) \subseteq P$

Explanations: Each hole in the pNet has a parameter p_j, expressing that this "parameterized hole" corresponds to as many actual arguments as necessary in a given instantiation of its parameter (we could have, without changing the expressiveness, several parameters per hole). In other words, the parameterized holes express *parameterized topologies* of processes synchronised by a given Net. Each parameterized synchronisation vector in the pNet expresses a synchronisation between some instances ($\{t\}_{t \in B_i}$) of some of the pNet holes ($I \subseteq J$). The hole parameters being part of the variables of the action algebra, they can be used in communication and synchronisation between the processes.

Fig. 6 is the parameterized version of the pNets for component B, in which the second hole (`B`) has a parameter `n`. The second synchronisation vector in the examples synchronises one (parameterized) action of the first hole `MC`, with an action (`?call(m,x)`) of the n^{th} instance of `B`. The comparison with the instantiated version in Fig. 4 shows clearly the benefits of parameterization, in term of compactness, and of generality. Note that this is still a very simplified and naive version of the pNet for B, the full semantics of GCM composite components will be given later.

A pNet by itself is stateless, but it has state variables that encode some notion of internal memory that can influence the synchronisation. pNets have the nice property that they can be easily represented graphically, e.g. using the Autograph editor [20].



where *B-param-Net* = $\langle P, pA_G, J, \tilde{p}_J, \tilde{O}_J, \overrightarrow{V} \rangle$ with:

P = {n, args, val, x}

pA_G = {?call(m,args), !resp(val), B[n].call(m,x), ...}

J = {MC, B}

p_{MC} = {}, p_B = {n}

O_{MC} = {?call(m,args), !resp(val), !call(m,x), ?resp(y)}

O_B = {?call(m,x), !resp(val), !call(foo()), !resp(z)}

$\overrightarrow{V}$ ={

⟨ ?call(m,args), ?call(m,args), - ⟩

⟨ B[n].call(m,x), !B(n).call(m,x), n&?call(m,x)⟩

... }

Fig. 6. Example of a pNet

Building hierarchical pNets. Once a pNet hierarchical system is built, you need operations to transform it, and, at least:

- a product operation for reducing a pNets hierarchy to a flat pLTS,
- a way of instantiating a parameterized pNet system with respect to a given domain for one or several of its parameters.

In [8], we gave the definition of pNets instantiation, and we defined the product operation only for fully instantiated systems. This is enough for instantiating a pNet system for some finite abstraction of the parameter domains, and building the global state-space of the system.

3.2 Model Generation for Active Objects

The first application of pNets that we have published was for defining the behavioural semantics of active objects of the ProActive library. In [21, 22] we presented a methodology for generating behavioural models for active objects (AOs), based on static analysis of the Java/ProActive code. The pNets model fits well in this context, and allows us to build compact models, with a natural relation to the code structure: we associate a hierarchical pNet to each active object of the application, and build a synchronisation network to represent the communication between them.

Fig. 7 illustrates the structure of the pNets expressing an asynchronous communication between two active objects. A method call to a remote activity goes through a proxy, that encodes the creation of the local future object, while the request goes to the remote request queue. Note that for each program point pp corresponding to a remote method call in the source code, a series of futures, indexed by a counter c, can be created. 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.

This method is composed of two steps: first the source code is analysed by classical compilation techniques, with a special attention to tracking references to remote objects in the code, and identifying remote method calls. This analysis produces a graph including the method call graph and some data-flow information. The second step consists in

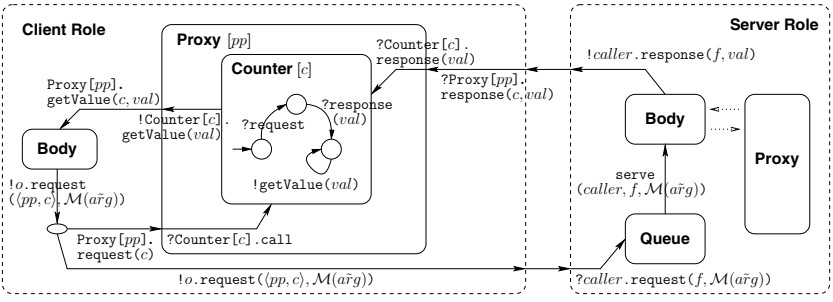


Fig. 7. Communication between two Active Objects

applying a set of structured operational semantics (SOS) rules to the graph, computing the states and transitions of the behavioural model.

The construction of the extended graphs by static analysis is technically difficult, and fundamentally imprecise. Imprecision comes from classical reasons (having only static information about variables, types, etc), but also from specific sources: it may not be decidable statically whether a variable references a local or a remote object. Furthermore, the middleware libraries include a lot of dynamic code generation, and the analysis would not be possible for Java code relying on introspection, classically used to manage some types of "dynamic topologies" in ProActive.

3.3 Model Generation for Hierarchical Components

Going from active objects to distributed and hierarchical components allows us to gain precision in the generated models. The most significant difference is that required interfaces are explicitly declared, and active objects are statically identified by components, so we always know whether a method call is local or remote. Moreover, the pNets's formalism expresses naturally the hierarchical structure of components, and will allow to scale up better, using compositional verification methods,

The pNet construction here may apply to any kind of hierarchical component model that features:

- Components with a set of interfaces and a content.
- Interfaces typed by a set of methods with their signature.
- Bindings between sibling subcomponents, or between a component and one of its subcomponent.
- Composite content composed of subcomponents, internal interfaces, and bindings.
- Empty content for primitive components.

We leave here undefined the code of a primitive component. It will depend on the framework, and will be used to generate a pLTS representing the primitive behaviour. We also leave undefined the data domains used for specifying indexes within the parameterized structure, and for building the arguments of the method calls.

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 global actions pA_G and hole sorts $\tilde{O}_J$ of the pNets are sets of actions of the form $[!|?]\, C_i.Itf.\mathcal{M}(\overrightarrow{arg})$ for invoking/serving a method $\mathcal{M}$ on the interface *Itf*.
- for each binding, and for each method in the signature of the source interface of the binding, it has two parameterized synchronisation vectors, one for sending the request, and one for receiving the response.

3.4 Hierarchical Components + Management Interfaces = Fractal

In the Fractal model, and in Fractal implementations, the ADL describes a static view of the architecture (used to build the initial component system through a *component factory*), and non-functional (NF) interfaces are used to control dynamically the evolution of the system. In this section we consider the core of the Fractal model, containing the hierarchical structure from the previous section, plus the basic non-functional interfaces and controllers, namely the Life-Cycle Controller (LF) and the Binding Controller (BC). We defined the behavioural semantics of Fractal applications in terms of pNets, giving the overall structure of the pNets encoding primitive and composite components, and the pLTS defining the LF and BC controllers.

A life controller pLTS (see Fig. 8) is attached to each component. Control actions (`start`/`stop`) are synchronised with the parent component and with all of its subcomponents. Status actions (`started`/`stopped`) are synchronised with the component's functional behaviour and with the BC, because the BC may only allow rebinding of interfaces when stopped.

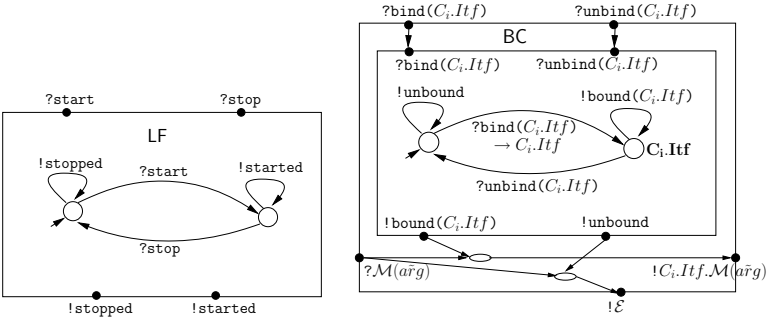


Fig. 8. pLTS of Fractal Life Cycle and Binding Controllers

A binding controller pLTS (see Fig. 8) is attached to each interface. Control actions (`bind`/`unbind`) are synchronised up to the higher level (Fractal defines a white-box definition for NF actions) and with the affected interface; status actions (`bound`/`unbound`) are used to allow method calls $\mathcal{M}(\tilde{arg})$, to forward the call to the appropriate bound interface and to signal errors. The latter is a distinguished action $\mathcal{E}(unbound, C, Itf)$, visible to the higher level of hierarchy, and triggered whenever a method call is performed over an unbound interface.

Fig. 9 sketches the structure of the synchronisation of a component with its subcomponents. In this drawing, the behaviour of subcomponents is represented by the box

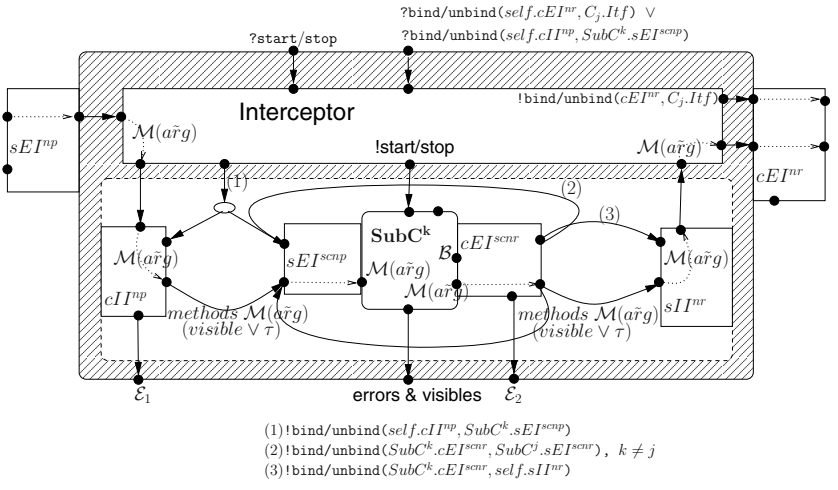


Fig. 9. Synchronisation pNet for a Fractal Composite Component

named *SubC*$^{\mathbf{k}}$. For each interface defined in the component's ADL description, a box encoding the behaviour of its internal (*cII* and *sII*) and external (*cEI* and *sEI*) views is incorporated. The dotted lines inside the boxes indicate a causality relation induced by the data flow through the box. Primitive components have a similar automaton without subcomponents and internal interfaces.

3.5 Model Generation for GCM

In Figure 10, we show the behavioural model of a GCM primitive component. There is a pLTS for dealing with the component's life-cycle (**LF**), and a pLTS for serving functional and non-functional requests (**Service**). The behavioural model for a composite component is an instance of the model of Figure 9, in which the interceptor itself is a primitive component.

Service implements the treatment of control requests. It interacts with the **LF** controller through the `!start` and `!stop` actions. The action `!start` fires the process representing the `runActivity()` method in the **Body**, and at the same time changes the LF state to "started". The `!stop` action is more complicated: it is sent by **Service** to the **Body**, but a running body may not be able to stop immediately upon reception of a stop request (because Java is non-interruptible). If the service policy of the component is the default FIFO, this `stop` request will be executed when all previous requests will be served. If the developer has specified his own `runActivity()` method, she/he has the responsibility for testing the presence of a `stop` request, and terminate the `runActivity()` method. At this point the `!stop` action will be transmitted to the **LF** controller, while the **Body** will be back in its initial state, ready for receiving a `!start` action.

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). It is always ready to perform any of the three actions numbered (1) to (3) in Fig. 10:

– (1) serve the first functional method obeying the selection mode;
– (2) serve a control method only at the head of the queue;
– (3) serve only control methods in FIFO order, bypassing the functional ones.

Depending on the state of the life-cycle controller, these actions may or may not synchronise with the Body and the Service pNets. This is encoded through the emission of the `!started` or `!stopped` actions by the LF pNet.

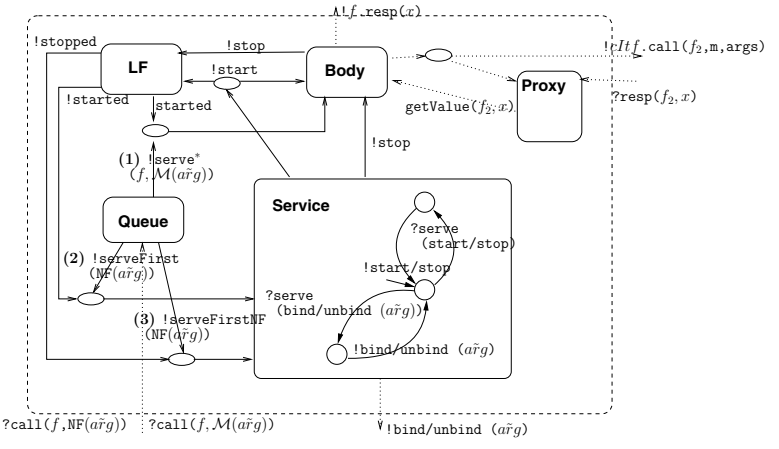


Fig. 10. Behavioural model of a primitive component

Modelling Collective Interfaces. Collective interfaces are responsible of distributing and gathering request calls and responses. Therefore, we provide a particular kind of proxy pLTS and N-ary synchronisation vectors encoding the control and data flow of these interfaces.

In Fig. 3, the multicast interface `MC` broadcasts request calls to all `B`'s subcomponents and gathers the results. We gave incomplete views of its pNet model, in Figures 4 and 6, and we show now its complete model in Figure 11. The proxy `Multicast(`f`)` pLTS is in charge of distributing the requests to all bound interfaces (in this case the server interfaces of `B`'s subcomponents). We use N-ary synchronisation vectors for broadcasting the call (`!call(args)`). This ensures that the call will be enqueued in every subcomponent at the same time. On the contrary, the response values of each component (`?resp(`*val*`)`) are sent back to the proxy individually and in any order. The proxy is in charge of gathering the result values in a vector. Later, when all results have arrived (guaranteed by the guard `[rep==N]`), it allows the component to access the result (`!getValue(`f`,`x`)`).

Modelling First-Call Futures. In Fig. 7 we depicted a simple proxy structure for ProActive futures. In GCM, futures can be transmitted in the parameters of a method call, or in the return value of a method call. In a naive approach, this requires knowing statically the flow of futures for each component because a future may have been created locally or by a third-party. This requires the analysis of the complete system. Instead, a better approach is to assign locally in each component an identifier f_{id} for

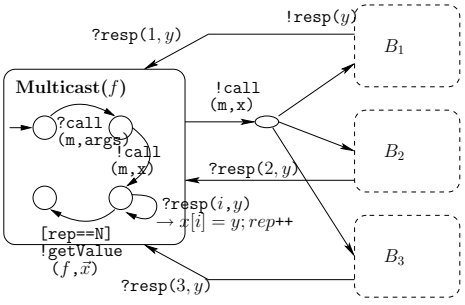


Fig. 11. Behavioural model of a multicast interface

each future, which permits the construction of behavioural models independently from the environment. Later, when the environment is known, the data-flow between components will determine which identifiers represent the same future object. At this point, these identifiers will be put in correspondence, and will be matched in the corresponding synchronisation vectors. This approach yields a compositional model.

In [23] we have shown the technical details of how to address different scenarios depending whether (i) a component transmits a locally created future; (ii) a component receives a future; and (iii) a component receives a future and retransmits it to a third-party. Here we define a new generic proxy that is able to deal with any combination of the 3 scenarios above. The proxy model has additional transitions w.r.t. the model presented in Figure 7 to allow futures to be transmitted. Figure 12 depicts this proxy[3].

When the local component is the creator of the future, the proxy starts by a transition `?call`. This allows the component to perform the remote remote call. In this case the proxy will wait for the `?response` transition to synchronise on the response value. Then there is a transition `!forward` for transmitting the future value to all components (if any) that may receive the future reference. Finally, the component body may access the content of the future through a `!getValue` transition.

Complementarily, if the local component did not create the future, the first transition of the proxy is a `?forward` which receives the value of the future. Afterward, the proxy behaves as in the previous case: it transmits the value to the remote components, and allows the component to access the future value.

Example: Sending a future created locally as a method call parameter. In Figure 12, the `Client` performs a method call $\mathcal{M}_1$ on `Server-A`, and creates a `Proxy(`f`)` for dealing with the result. Then the `Client` sends the future to a third activity (`Server-B`) in the parameter of the method $\mathcal{M}_2(f)$ (this call should eventually create another future f_2, but we have omitted it for simplicity).

[3] In this modelisation, we have an unbounded number of proxy instances, that live forever, and don't need to be terminated/destroyed. In the implementation, we may want to be more efficient: based on static analysis, the implementation can decide that some futures have a limited life-time, and that they can be destroyed or recycled at some point. Then we may want to prove correctness of such an optimisation.

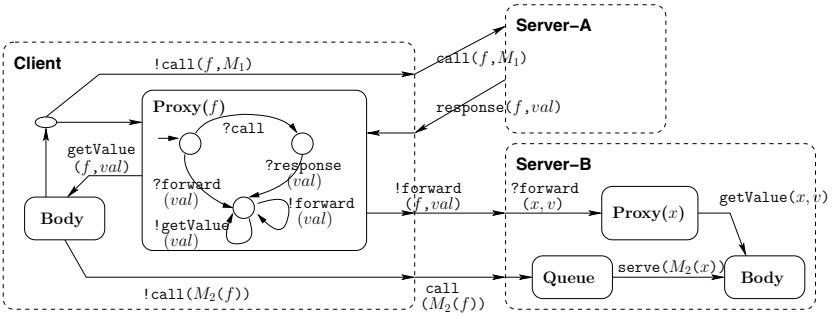


Fig. 12. Model for sending a future created locally as a method call parameter

From `Server-B`'s point of view, there is no way of knowing if a parameter is (or contains) a future, so every parameter in a method call must be considered as a potential future. `Server-B` includes, therefore, a proxy for dealing with the parameter x of the method call M_2.

This example concludes the construction of pNets models for GCM components, incorporating non-functional controllers, request queues, future proxies, and NxM communication. In the current implementation, described in the next sections, the NxM communication and the proxies for first class futures are not yet supported.

4 VerCors: A Toolset for Specification and Verification

In this section, we report on the tool developments ongoing within our VerCors platform, implementing the behaviour model generation explained in the first half of this paper. We start with a description of the current and middle term functionalities of the platform, and we explain briefly the software tools used for the construction of the platform. Then we give more details on the graphical editors, on the model generation tool, and the model instantiation tools. Finally, we discuss some pragmatic aspects of various verification strategies for using the tools, and give some figures on typical case-studies.

4.1 Vercors Architecture

Fig. 13 sketches the architecture of VerCors. This toolset is available as free software, from our web site [9]. The platform has two goals: the verification of designs, and the generation of safe-by-construction code. In the following description of the VerCors modules, we shall indicate which functionalities are already available in the distribution (V0.2, spring 2009), and which are still under construction.

Front-End. VCE (for Vercors Component Editor) is our graphical component editor for designing components. It provides diagrams for defining the component architecture (see Section 4.3), and diagrams for defining the component behaviour (see Section 4.4). The latter is not yet available in V0.2. The Java Distributed Components specification language (JDC) is a textual language more expressive than our graphical diagrams, but is not yet implemented. It has been described in [10, 24].

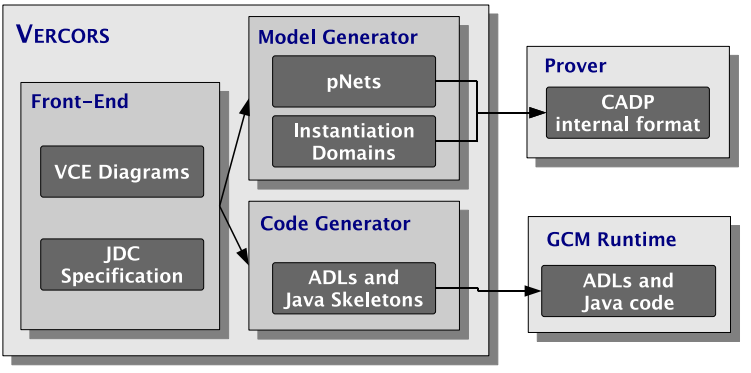


Fig. 13. The VerCors toolset

Model Generator. The model generator is the kernel of the platform. It is fed with specifications given by VCE diagrams or JDC specifications. It includes tools for data abstraction (from user-defined classes in JDC to Simple Types in pNets), tools for building the parameterized models from the specifications, and tools for manipulating and instantiating pNets (see section 4.5).

Code Generator. Another central part of the platform will be the code generator that is not (yet) currently developed. We will generate code capable of running under the standard GCM specification. It has an architecture definition based on the GCM ADL and Java code based on GCM / ProActive framework. The latter must be refined by the user by filling-in the business code.

External Tools. Externally to the platform, we interact with model-checking engines and with the GCM runtime. For now, VerCors uses the CADP toolset [25] for distributed state-space generation, hierarchical minimization, on-the-fly verification, and equivalence checking (strong/weak bisimulation). The connection with CADP is done through various textual input formats, that we generate from (fully instantiated) pNet models. A better approach would be to use a more generic and standardized intermediate format, like the FIACRE format [26], that would allow us to represent directly many (parameterized) constructs from the pNet model.

Verification is done by verifying regular μ-calculus formula encoding the user requirements. In the future, we would like to specify these properties within JDC, which would be subject to the same abstractions, and finally be translated into regular μ-calculus formula. We also plan to use other state-of-the-art provers, and in particular apply so-called "infinite system" provers to deal directly with certain types of parameterized systems.

4.2 Building Tools Using Eclipse Meta-modelling Framework

From a practical point of view, VCE consists of graphical editors for specifying the architecture and the behaviour of distributed components. It is built as an Eclipse plug-in based on EMF and GEF.

We use two similar meta-modelling frameworks, namely Topcased [27] and GMF. EMF plays the role of the *domain model* whereas Topcased and GMF provide graphical editors on top of the domain model. Unfortunately, Topcased is slowing down the development of their meta-modelling framework and future support is uncertain. Therefore, our early work on the architectural editor is generated by Topcased, but our more recent work on the behavioural editor is generated by GMF.

Model validation is based on OCL (Object Constraint Language) [28] rules that validate instances of the meta-model, and Java code that checks interface compatibility. There are a minimum set of invariants that every model must hold. Complementary, an additional set of rules cope with particular GCM implementations. All errors in the user models are reported in the Eclipse environment.

There is also compatibility with GCM ADL files. VCE is able to import and export GCM ADL files, though this is limited to functional components since there is no standard definition of NF components in the GCM ADL.

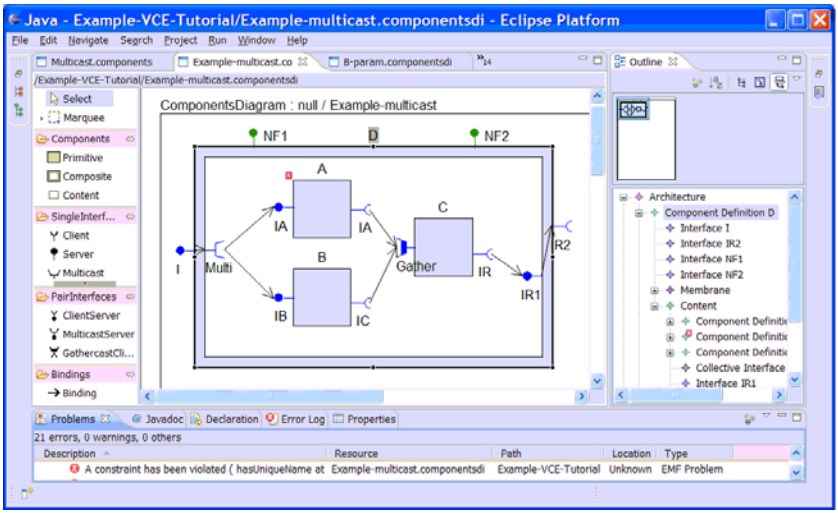


Fig. 14. Vercors Component Editor

4.3 Graphical Diagrams for Component Architecture

The kernel of the graphical language is a meta-model that reflects the GCM component structure. As these graphical constructions have already been used throughout this paper, we will only comment here on the main design choices that we have made.

At the top-level, the designer defines the root component that sets the services to be provided and required by the application to the environment. A component has a *content* that implements the business code, and a *membrane* that contains the non-functional code.

Components in the content are called *functional* components and those in the membrane are called *non-functional* (NF) components. The content is represented as a white

rectangle inside the component, and the membrane is the grey area that surrounds the content. Nevertheless, the content of primitive components is not depicted; therefore, primitive components are distinguished as grey rectangles. We colour blue the "usual" functional interfaces, and green the NF interfaces.

Interface icons are inspired by the ones used in UML component diagrams. Server interfaces are drawn as filled circles (e.g. interfaces `I`, `IA`, ... in Figure 14), and client interfaces as semi-circles (e.g. interfaces `IC`, `IR`, ...). GCM's *collective interfaces* are not defined in UML and hence we adopted our own icons. Figure 14 also shows the icons we provide for *multicast* and *gathercast* interfaces, labelled `Multi` and `Gather` respectively. In the example, the interface `Multi` broadcasts incoming requests to components `A` and `B`, and the interface `Gather` gathers and synchronises requests coming from interfaces `IC` of components `A` and `B` towards the component `C`.

4.4 Diagrams for Behaviour Specification

The diagrams for behaviour specification have been defined in [29], but the diagram editors are not yet available in the toolset. They are based on a variant of UML 2 State Machine diagrams, with a number of State Machines used to specify respectively: the component service policy, each service method and each local method, the interface policies, etc.

4.5 Model Generation

The role of the ADL2N tool is to:

- build an abstract version of the component system, in which the user-defined Java classes used for the parameter domains are abstracted by some Simple Types from the pNets library.
- use the behaviour semantics defined in sections 3.3 to 3.5 to build the pNet model for each piece of the system.

The first step of the model generation deals with data abstraction: data types in a JDC specification are standard, user-defined Java classes, but they must be mapped to Simple Types before generating the behavioural models and running the verification tools. The result is an abstract specification with the same structure than the initial ADL.

In practice the user of ADL2N uses a GUI to specify at the same time the methods that will be visible, the arguments that are significant for the proofs, and finite domains for these arguments. This is shown in Fig. 15. Here some tool guidance would be very helpful to reduce the amount of user input required, and to guarantee the coherency of the abstraction with the dataflow within the system. This kind of guidance is not yet available in the toolset.

Such an Abstract Specification will then be given as input to the model generator. This tool builds a model in terms of pNets, including all necessary controllers for non-functional and asynchronous capabilities of the components. The only missing part is the functional behaviour (Body) of primitive components for which ADL2N only defines their sorts.

The second usage of the abstraction module of ADL2N is to specify a *finite* abstraction of the parameters domains (from Simple Types to finite Simple Types), so that the

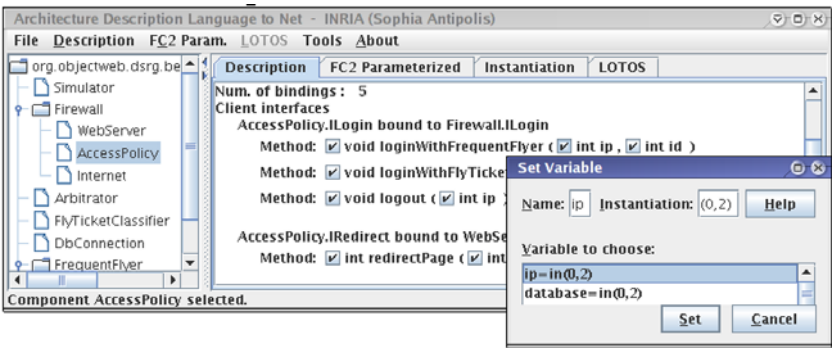


Fig. 15. Screenshot of ADL2N

final pNet system is finite, and suitable for analysis with finite-state model-checkers. In practice ADL2N produces two files, one file with the parameterized system, the other file with the definitions of the finite instantiations for the parameter domains.

pNets instantiations and export formats. The textual notation we use currently in the platform to encode pNets is called FC2 [30]. We provide two tools, FC2Instantiate and FC2Exp [31], that create finite instantiations of the models and transform the files into the input formats of CADP, namely BCG for transition systems, and Exp for synchronisation vectors [32].

4.6 Model-Checking: Engineering, Pragmatic Complexity

Having produced our models in a structured and hierarchical format allows us to use many pragmatic strategies to master as much as possible the state-space complexity of model-checking. The main tool is compositionality: as we use a bisimulation-based verification toolset, it is essential that each intermediate subsystem is reduced (by branching or weak minimization) before being synchronized with others. If we are careful to reduce as much as possible the visibility of actions, then state-space explosion can be contained (to some extent) within the model of composite components. Additionally, a number of advanced features of the CADP toolset can help us to fight state-explosion, and to scale up. Typically, we can build the state-space at each level of the hierarchy using the distributed state-space generation of CADP, including on-the-fly hiding and tau-reduction, but also behaviour generation constrained by the environment. Then the minimization has to take place on a single machine, because the bisimulation engine is not implemented in a distributed way. And the next cycle of construction can be distributed again... This way your state-space construction can scale up to any system in which the largest intermediate structure will be in the range of 10^8 states. The model-checker engine itself has an experimental version working in a distributed fashion.

Using this kind of strategy, we have done some middle-size case studies, including for example the Common Component Modeling Example (CoCoME, [33]). This is a system of 17 components structured in 5 levels of hierarchy, with more than 10 data parameters, and some broadcast communication. We have treated this case using the

Fractal model generation (3.4), with very small abstract domains for the variables (typically 2 or 3 values). The brute force state space for this would be approximately 2.10^8, while the biggest intermediate structure that we generate is lower than 10000 states. We have shown in [33] a number of properties and problems verified on this model.

Such models can be used to check the satisfiability of safety or liveness formulas in branching time logics, or to check the bisimulation equivalence with respect to an abstract specification. In practice, we want to provide non-expert users with simple "press button" verification functions. This is easy for some families of reachability properties, like correct termination of deployment, or occurrence of some predefined sets of error actions. Deadlock detection is also a popular "push button" function, but explaining to the user the reasons of a deadlock can be challenging; it often involves some "missed synchronisation", that may be difficult to show, especially in presence of abstraction and instantiation.

The type of properties we can check on our models are more versatile than in most approaches, because we do not only encode the usual functional interactions between the components, but also their reconfiguration operations. So we can prove properties of applications in which one would change bindings, or remove and update subcomponents, while the rest of the system keeps running. This kind of properties typically depends on the behaviour of the system parts, and is not a general property of the middleware.

5 Conclusion and Perspectives

In this paper we have presented the models and tools we have been implementing to assist the development of Grid component-based applications. The approach is based on the modelling of the component behaviour using parameterized networks of automata. In addition, we have presented tools that generate these models, and tools for the specification of the component system.

This paper makes a step forward towards the verification of Grid applications. It provides novel models for multicast interfaces and generic proxies for transmitting futures. Moreover, one of the strong original aspects of this work is the focus put on non-functional properties, and the results we provide on the interleaving between functional and non-functional concerns. Thus, the programmer should be able to prove the correct behaviour of his distributed component system in presence of evolution (or reconfiguration) of the system.

We are currently developing additional tools in the VerCors platform to support our methodology. This includes the front-ends for textual and graphical specification languages, a tool for helping the user to build correct abstractions, and tools for providing readable explanations of the provers diagnostics.

Finally, we have presented techniques to master state-space explosion. The key aspect is the use of compositionality to reduce the system at each level of hierarchy. Nevertheless, in some cases, particularly when queues are unbounded, state-space explosion is inevitable when using explicit-state model-checkers. Therefore, our latest work focuses on the development of an infinite-state model-checker that verifies automata endowed with unbounded FIFO queues.

References

[1] Szyperski, C.: Component Software, 2nd edn. Addison-Wesley, Reading (2002)

[2] Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J.-B.: An open component model and its support in java. In: Crnković, I., Stafford, J.A., Schmidt, H.W., Wallnau, K. (eds.) CBSE 2004. LNCS, vol. 3054, pp. 7–22. Springer, Heidelberg (2004)

[3] CoreGRID, Programming Model Institute: Basic features of the grid component model (assessed). Technical report, CoreGRID, Programming Model Virtual Institute, Deliverable D.PM.04 (2006), http://www.coregrid.net/mambo/images/stories/Deliverables/d.pm.04.pdf

[4] Oquendo, F.: π-ADL: An Architecture Description Language based on the Higher Order Typed π-Calculus for Specifying Dynamic and Mobile Software Architectures. ACM Software Engineering Notes 26(3) (2004)

[5] Groote, J., Mathijssen, A., Reniers, M., Usenko, Y., van Weerdenburg, M.: The Formal Specification Language mCRL2. In: Proc. Methods for Modelling Software Systems (2007)

[6] Poizat, P., Royer, J.-C., Salaün, G.: Bounded Analysis and Decomposition for Behavioural Descriptions of Components. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. LNCS, vol. 4037, pp. 33–47. Springer, Heidelberg (2006)

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

[8] Barros, T., Boulifa, R., Cansado, A., Henrio, L., Madelaine, E.: Behavioural models for distributed Fractal components. Annals of Telecommunications 64(1-2) (January 2009); also Research Report INRIA RR-6491.

[9] OASIS team: VerCors: a Specification and Verification Platform for Distributed Applications (2007-2009), http://www-sop.inria.fr/oasis/index.php?page=vercors

[10] Cansado, A., Henrio, L., Madelaine, E., Valenzuela, P.: Unifying architectural and behavioural specifications of distributed components. In: International Workshop on Formal Aspects of Component Software (FACS 2008), Malaga, Electronic Notes in Theoretical Computer Science (ENTCS) (September 2008)

[11] Caromel, D., Henrio, L.: A Theory of Distributed Objects. Springer, Heidelberg (2005)

[12] Caromel, D., Henrio, L.: Asynchonous distributed components: Concurrency and determinacy. In: Proceedings of the IFIP International Conference on Theoretical Computer Science 2006 (IFIP TCS 2006), Santiago, Chile, August 2006. Springer Science (2006); 19th IFIP World Computer Congress

[13] Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J.-B.: An open component model and its support in java. In: Crnković, I., Stafford, J.A., Schmidt, H.W., Wallnau, K. (eds.) CBSE 2004. LNCS, vol. 3054, pp. 7–22. Springer, Heidelberg (2004)

[14] Seinturier, L., Pessemier, N., Coupaye, T.: AOKell: an Aspect-Oriented Implementation of the Fractal Specifications (2005), http://www.lifl.fr/~seinturi/aokell/javadoc/overview.html

[15] European Telecommunication Standards Institute, http://portal.etsi.org

[16] Caromel, D., Delbé, C., di Costanzo, A., Leyton, M.: ProActive: an integrated platform for programming and running applications on grids and P2P systems. Computational Methods in Science and Technology 12(1), 69–77 (2006)

[17] Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)

[18] Arnold, A.: Finite transition systems. Semantics of communicating sytems. Prentice-Hall, Englewood Cliffs (1994)

[19] Lin, H.: Symbolic transition graph with assignment. In: Sassone, V., Montanari, U. (eds.) CONCUR 1996. LNCS, vol. 1119. Springer, Heidelberg (1996)

[20] Madelaine, E.: Verification tools from the CONCUR project. EATCS Bull. 47 (1992)
[21] Barros, T., Boulifa, R., Madelaine, E.: Parameterized models for distributed java objects. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004, Madrid. LNCS, vol. 3235, pp. 43–60. Springer, Heidelberg (2004)
[22] Boulifa, R.: Génération de modèles comportementaux des applications réparties. PhD thesis, University of Nice - Sophia Antipolis – UFR Sciences (December 2004)
[23] Cansado, A., Henrio, L., Madelaine, E.: Transparent first-class futures and distributed component. In: International Workshop on Formal Aspects of Component Software (FACS 2008), Malaga, Electronic Notes in Theoretical Computer Science, ENTCS (September 2008)
[24] Cansado, A.: Formal Specification and Verification of Distributed Component Systems. PhD thesis, Université de Nice - Sophia Antipolis – UFR Sciences (December 2008)
[25] Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In: CAV (2007)
[26] Berthomieu, B., Bodeveix, J.P., Filali, M., Garavel, H., Lang, F., Peres, F., Saad, R., Stoecker, J., Fran, C.V.: The Syntax and Semantics of FIACRE V2.0. Technical report (Feburary 2009)
[27] Pontisso, N., Chemouil, D.: Topcased combining formal methods with model-driven engineering. In: ASE, pp. 359–360. IEEE Computer Society, Los Alamitos (2006)
[28] Object Management Group: UML 2.0 Object Constraint Language (OCL) Specification. formal/03-10-14 edn, version 2.0 (2003)
[29] Ahumada, S., Apvrille, L., Barros, T., Cansado, A., Madelaine, E., Salageanu, E.: Specifying Fractal and GCM Components With UML. In: Proc. of the XXVI International Conference of the Chilean Computer Science Society (SCCC 2007), Iquique, Chile, Nov 2007, IEEE, Los Alamitos (2007)
[30] Ressouche, A., de Simone, R., Bouali, A., Roy, V.: The FC2Tool user manuel (1994), http://www-sop.inria.fr/meije/verification/
[31] Barros, T.: Formal specification and verification of distributed component systems. PhD thesis, University of Nice - Sophia Antipolis (November 2005)
[32] Lang, F.: Exp.Open 2.0: A flexible tool integrating partial order, compositional, and on-the-fly verification methods. In: Romijn, J.M.T., Smith, G.P., van de Pol, J. (eds.) IFM 2005. LNCS, vol. 3771, pp. 70–88. Springer, Heidelberg (2005)
[33] Rausch, A., Reussner, R., Mirandola, R., Plášil, F.: The Common Component Modeling Example. LNCS, vol. 5153. Springer, Heidelberg (2008)