Verification of Distributed Hierarchical Components

Tomás Barros[1] and Ludovic Henrio[2] and Eric Madelaine[1]

[1] *INRIA Sophia Antipolis, CNRS - I3S - Univ. Nice Sophia Antipolis 2004, Route des Lucioles, BP 93, F-06902 Sophia-Antipolis Cedex - France*
[2] *Univ. of Wesminster, Watford Rd Northwick park, Harrow, HA1 3TP, UK*
Email :First.Last@sophia.inria.fr

Abstract

Components allow to design applications in a modular way by enforcing a strong separation of concerns. In distributed systems this separation of concerns have to be composed with distribution of controls due to asynchrony. This article relies on Fractive, an implementation of the Fractal component model allowing to unify the notion of components with the notion of activity.

This article shows how to build automatically the behaviour of a distributed component system. Starting from the functional specification of primitive components, we generate a specification of a system of components, their asynchronous communications, and their control. We then show how to use such a specification to verify properties specific to components, reconfigurations, or asynchrony.

Key words: Hierarchical components, behavioural specification, distribution, asynchrony.

1 Introduction

Component programming has emerged as a programming methodology ensuring both re-usability and composability. Components inherit from a long experience about modules, objects and interfaces.

The Fractal [4] component model provides hierarchical composition for a better structure, and specification of control interfaces for dynamic management. The various control interfaces allow the execution control of a

[0] This research work is carried out under the ACI Securité FIACRE funded by the french government, and under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265) and under the associated team OSCAR funded by INRIA and University of Chile

component and its dynamic evolution: plugging and unplugging components dynamically provide adaptability and maintenance. Particularly, distributed components have to feature dynamic adaptation to their environment.

This article aims at a framework for the behavioural specification and verification of *distributed*, *hierarchical*, *asynchronous*, and *dynamically reconfigurable* components built based on the Fractal specification. The challenge that is addressed is to build a formal framework ensuring both correct composition at deployment (design and implementation), and safe dynamic changes or reconfigurations (maintenance and adaptation). Therefore the intended user of our framework is the application developer in charge of those tasks. This framework should hide as much as possible the complexity of the verification process, and be as automatic as possible.

Some early work on behaviour specification of components, such as Wright or Darwin, are based on process algebras. In Sofa [11] components have a *frame* (specification) and *architecture* (implementation) protocols, and verification is done through a trace language inclusion of the *architecture* within the target *frame*. In a different flavor, the work of Carrez et al on behavioural typing of components [5] gives a *sound assembly* and compatibility definition which ensures correctness of the composition. But most of the recent developments on correct components, e.g. in the Mobj and Eureka projects, aim at reactive systems and do not consider asynchronous models. To our knowledge, no other work consider the interplay of component management with the user-defined *functional behaviour*.

Our approach is to give behavioural specifications of the components in the form of hierarchical synchronised transition systems. The models for the functional behaviour of basic components may be derived, as described in [1], from automatic analysis of source code, or expressed in a dedicated specification language. Control (or non-functional) behaviour is automatically incorporated within a controller built from the component's description. The semantics of a component is then computed as a product of the LTSs of its sub-components with the controller of the composite. This system can be checked against requirements expressed as a set of temporal logic formulas, or again as an LTS.

The next section reviews Fractal and its distributed implementation Fractive [3]. In section 3 we show how to generate the behavioural model of primitive and composite components. In section 4 we explain how the user specifies both the functional behaviour of primitive components and the control features of their composition. Finally Section 5 shows how our tools can be used to prove behavioural properties on a system of distributed components.

2 Context

We focus on component based systems built using *Fractive*. Fractive is a Fractal implementation using the *ProActive* middle-ware [7]. Thus, it provides

a component model having the same features as *ProActive*, the most important being asynchronous method calls, absence of shared memory, user-definable service policy, and transparency versus distribution and migration.

2.1 Fractal

A Fractal component is formed out of two parts: a *controller* (or membrane) and a *content*. Fig. 1 shows an example of a Fractal component system.

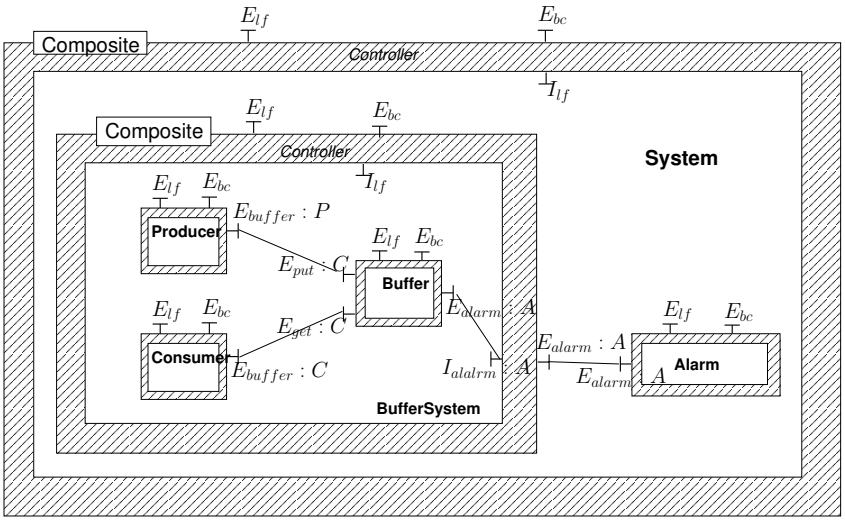

Fig. 1. A Fractal component

The controller of a component can have *external* interfaces (e.g., E in Fig. 1) and *internal* ones (e.g., I in Fig. 1). A component can interact with its environment through *operations* at its external interfaces, while internal interfaces are accessible only from the component's sub-components.

Interfaces can be of two sorts: *client* and *server*. A server interface receives method invocations while a client interface emits method calls. A *functional* interface provides or requires functionalities of a component, while a *control* interface corresponds to a management feature over the component architecture. Fractal defines 4 types of control interfaces: *binding control*, to bind/unbind the client interfaces (e.g. E_{bc} in Fig. 1); *life cycle control*, to stop and start the component (e.g. E_{lf} in Fig. 1); *content management* to add/remove/update sub-components, and *attribute control* to get/set internal attributes. This paper focuses on the first two. A component can perform content and binding operations only when *stopped* and can emit invocations only when *started*.

2.2 ProActive

ProActive is a pure Java implementation of distributed active objects with asynchronous remote method calls and replies by means of future references. A distributed application built using *ProActive* is composed of several *activities*, each one having a distinguished entry point, the *active object*, accessible from

anywhere. All the other objects of an activity (called *passive objects*) can not be referenced directly from outside. Each activity owns its own and unique service thread and the programmer decides the order in which requests are served by overloading the `runActive` method (entry point of the activity). The method calls to active objects behave as follows:

(i) When an object performs a method call to an active object (e.g., $y = \mathbf{O}_B.m(\boldsymbol{x})$), the call is stored in the request queue of the called object and a future reference is created and returned (y references f). A future reference encodes the promised return of an asynchronous method call.

(ii) At some point, the called activity decides to serve the request. The request is taken from the queue and the method executed.

(iii) Once the method finishes, its result is updated, i.e. the future reference (f) is replaced with the concrete method result (`value of` y).

When a thread tries to access a future before it has been updated, it is blocked until the update takes place (*wait-by-necessity*). The ASP calculus [6] has been defined to provide a computation model for *ProActive*.

2.3 Fractive

Fractive is the Fractal implementation using *ProActive*. Some features are left unspecified in the Fractal definition, and may be set by a particular Fractal implementation, or left to be specified at user level. Fractive makes the choice that the start/stop operations are recursive, i.e. they affect the component and each one of its sub-components, in a top-down order.

2.3.1 Primitive Components

A primitive component in Fractive is made from one activity whose active object implements the provided interfaces. Both, functional and control requests are dropped in the request queue of the active object. A Fractive primitive behaves as follows:

(i) When stopped, only control requests are served.

(ii) Start a primitive component means run the `RunActive` method of its active object

(iii) Stop a primitive component means exit from the `RunActive` method. Since active objects are non-preemptive, the exit from the `RunActive` method can not be forced: stop requests are signalled by setting the local variable `isActive` to false; then, the `RunActive` method should eventually end its execution.

2.3.2 Composites

Fractive 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 any Fractive composite is shown in Figure 2.

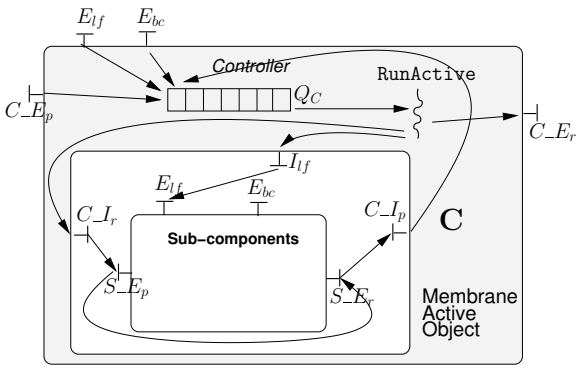

Fig. 2. Fractive composite component

The service thread serves the requests in FIFO order but only serves the control requests when the composite is stopped. As a consequence, a *stopped* composite will not emit functional calls on its required interfaces, even if its sub-components are active and send requests to its internal interfaces. Serving a functional request on an internal provided interface means forwarding the call to the corresponding external required interface of the composite. Serving a functional request on an external provided interface consists in forwarding the call to the corresponding internal required interface of the composite.

3 Behavioural Models

The core of our work consists in synthesizing a behavioural model of each component, in the form of set of synchronised labelled transition systems (LTSs). The formal model has been defined in [1] where we have shown how to build the behaviour of ProActive activities ; this corresponds exactly to the functional part of the behaviour of primitive components in Fractive.

Using the same formal model, [2] shows how to generate the control part of Fractal components. This article uses a similar approach for supporting Fractive. Due to size limits, we cannot recall in detail the construction of the LTSs corresponding to the Fractal control operations, only the elements required for an independant reading of the paper are presented below.

Given the functional behaviour of a primitive component, or of the sub-components of a composite, we extract from its architectural description the information required to generate LTSs encoding its control features (life-cycle and binding). The semantics of a component is then computed as the synchronised product of all those part, and is named the component's *controller* automaton.

The construction is done bottom-up through the hierarchy. At each level, i.e. for each composite, a deployment phase is applied. The deployment is

a sequence of control operations, expressed by an automaton, ending with a distinguished successful action $\surd$. A successful deployment is verified by the reachability analysis of the $\surd$ action on the automaton obtained by the synchronisation product of the component's controller and its deployment.

As in [2], we define the *static* automaton of a component as being the synchronisation product of the *controller* automaton with the deployment automaton, hiding control actions, forbidding any further reconfiguration, and minimised modulo weak bisimulation. When one is not interested in reconfigurations, the static automaton becomes the LTS encoding the behaviour of this sub-component at the next level of the hierarchy.

Fig. 3 shows the *controller* for a Fractive component at any level of the hierarchy.

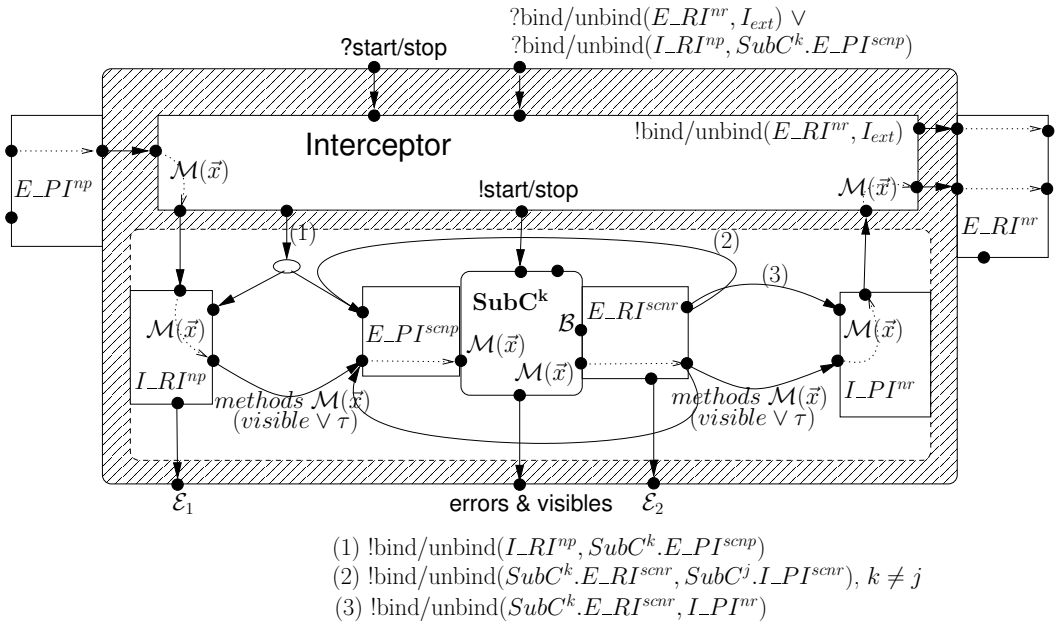

Fig. 3. Component behaviour model

In the figure, the behaviour of sub-components (i.e. their static LTS) is represented by the box named **SubC$^{\mathbf{k}}$**. For each interface defined in the component's ADL description, a box encoding the behaviour of its internal (I_PI and I_RI) and external (E_PI and E_RI) views are incorporated. The treatment of Fractive method calls is encoded in the box named Interceptor which we detail later. The doted edges inside the boxes indicate a causality relation induced by the data flow through the box.

The behaviour of the interfaces includes functional (method calls $\mathcal{M}(\overrightarrow{x})$) and non-functional (control) aspects, as well as the detection of errors ($\mathcal{E}_1$ and $\mathcal{E}_2$) such as the use of an unbound interface. These errors are made visible at the higher level of the hierarchy. For instance in Fig. 4 is shown the details of I_RI^{np} which includes the creation of an error event when a method is called on an unbound interface.

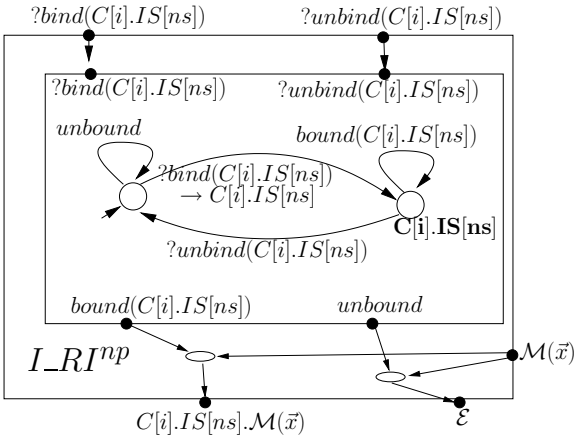

Fig. 4. Internal interface box detail

Note that we put the external interface automaton of a component in the next level of the hierarchy. This enables us to calculate the *controller* automaton of a component before knowing its environment. Thus, all the properties not involving external interfaces can be verified in a fully compositional manner.

3.1 Modelling the Primitives

Figure 5 shows the principle of asynchronous communication between two Fractive primitive components.

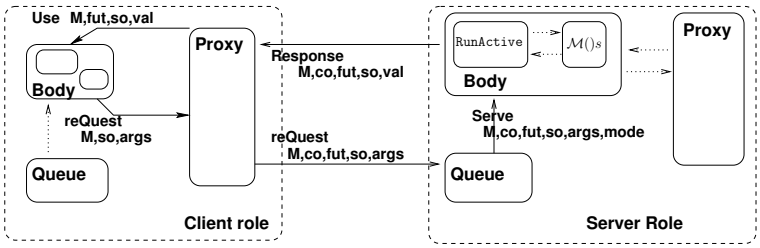

Fig. 5. Communication between two Activities

In the model (Fig. 5), 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 a reference to the future, together with a deep copy of the method's arguments, because there is no sharing between remote activities. Later, the request may eventually be served, and its result value will be sent back to the future reference.

The **Body** box in the figure is itself a synchronisation network made from the synchronisation product of the `RunActive` method's LTS with the behaviour of each method as described in [1]. The **Queue** box, additionally to requests reception, encodes the different primitives (used in the body code) provided in the *ProActive* API for serving the methods in the queue.

In the model of a Fractive primitive component we enrich the controller

of the active object by adding two extra boxes, **LF** and **NewServe** (which correspond to the Interceptor in Fig. 3) as shown in Fig. 6. The **body** box is the only part that is not generated automatically.

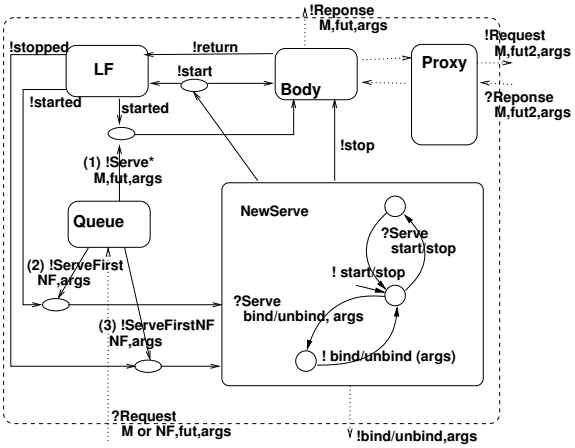

Fig. 6. Behaviour model for a Fractive primitive

NewServe implements the treatment of control requests. "start" fires the `RunActive` method (transition) in **body**. "stop" triggers the `!stop` synchronisation with **body** (Fig.6). This synchronisation should eventually lead to the termination of the `RunActive` method (`!return` synchronisation). In the Fractive implementation, this is done through setting the state variable `isActive` to false, which should eventually cause the `RunActive` method to finish, only then the component is considered to be stopped.

The Queue box can perform three actions: (1) serve the first functional method corresponding to the `Serve` API primitive used in the body code, (2) serve a control method only in the head of the queue, and (3) serve only control methods in FIFO order, bypassing the functional ones.

3.2 Modelling the Composites

A composite membrane in Fractive is an active object. When started, it serves functional or control methods in FIFO order, forwarding method calls between internal and external functional interfaces. When stopped it serves only control requests.

The membrane active object is created based on the composite description (given by the ADL). This membrane corresponds to the Interceptor box in Fig. 3. Note that the future references (**proxy** box in Fig. 7) are updated in a chain following the membranes from the primitive serving the method to the caller primitive. Since the method calls include the reference of the future in the arguments, future updates can be addressed directly to the caller immediately before in the chain. Consequently, like in the implementation, the future update is not affected because of rebinding or the life-cycle status of the components.

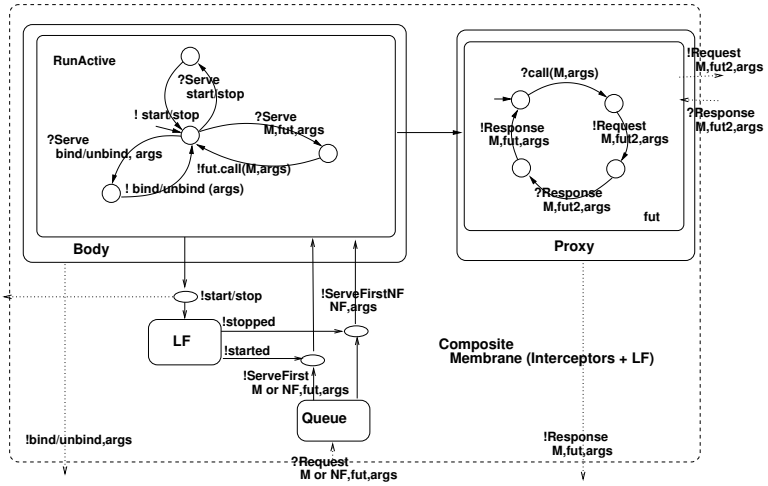

Fig. 7. Behaviour of a composite membrane

3.3 Building the Global Behaviour

The next step is to build a global model for the component. This "global" behaviour construction is compositional in the sense that each level of hierarchy can be studied independently, relying on some abstraction of the subcomponents behaviours. In practice, the abstract model of a subcomponent can be defined by its formal specification, or computed recursively from analysis of its ADL and its code.

As in our previous work [1,2], we build finite abstraction of our models using finite instantiations of the data values of parameters, before computing any synchronous product. Whenever the checking tools allow it, this instantiation and the corresponding state space generation is done on the fly during the proof. This data instantiation is interpreted as a partition of the data domains and induces an abstract interpretation of the parameterized LTS. The instantiation will also be chosen with respect to the values occurring in the properties we are interested in.

4 The User View

The models for the non-functional aspects described in this paper are built automatically. The user only has to provide the architecture through the Fractal ADL and the functional behaviour of the primitive components.

4.1 Looking at one Example

We come back to our example from Figure 1. It shows, as a hierarchical component system, the classical problem of a bound buffer with one consumer and one producer. The consumer consumes one element at a time while the the producer may feed the buffer with an arbitrary quantity of elements in one action. Additionally the buffer emits an alarm through its interface I_{alarm},

when the buffer is full.

The user may describe the system topology using the Fractal Architecture Definition Language (ADL). Fractive uses the default concrete syntax for this ADL based on XML. The XML file describing **System** is shown in Fig. 8.

System.fractal

```
1  <?xml version="1.0" encoding="ISO-8859-1" ?>
2  <!DOCTYPE .... >
3
4  <definition name="components.System">
5
6     <component name="BufferSystem"
7           definition="components.BufferSystem(3)">
8        <interface name="alarm" role="client"
9              signature="components.AlarmInterface"/>
10    </component>
11
12    <component name="Alarm">
13       <interface name="alarm" role="server"
14             signature="components.AlarmInterface"/>
15       <content class="components.Alarm">
16          <behaviour file="AlarmBehav"
17                format="FC2Param"/>
18       </content>
19    </component>
20
21    <binding client="BufferSystem.alarm"
22          server="Alarm.alarm"/>
23 </definition>
```

Fig. 8. System ADL

Buffer.lotos

```
process BUFFER[NOACTIVE, SERVE_GET, GET_REP,
    PUT, ALARM](stock:Nat, bound:Nat): exit :=

  PUT ?X:Nat [X <= (bound - stock)];
     BUFFER[NOACTIVE, SERVE_GET, GET_REP,
        PUT, ALARM] (stock+X,bound)
  []
  [stock > 0] -> SERVE_GET?C:Cons; GET_REP!C;
     BUFFER[NOACTIVE, SERVE_GET, GET_REP,
        PUT, ALARM](stock-1,bound)
  []
  [stock == bound] -> ALARM;
     BUFFER[NOACTIVE, SERVE_GET, GET_REP,
        PUT, ALARM](stock,bound)
  []
  NOACTIVE; exit
endproc
```

Fig. 9. Buffer behaviour

The XML description shown in Fig. 8 specifies that the system is composed of the composite **BufferSystem** (line 6), itself described in a separate file (`components/BufferSystem.fractal`), and the primitive **Alarm**, which implementation is the Java class `components.Alarm` (line 15). **BufferSystem** receives as construction parameter the maximal size of the buffer (3 in our example, line 7) and requires an interface named `alarm` of type `components.AlarmInterface` (lines 8,9). **Alarm** provides an interface `alarm` of type `components.AlarmInterface` (lines 13,14). The `behaviour` tag (line 16) points to a file containing the behaviour of `alarm` in LTS form.

Finally, at lines 21, 22, the ADL defines that upon deployment, the interface `alarm` of **BufferSystem** should be bound to the interface `alarm` of **Alarm**.

4.2 Automatic Construction

Our set of tools includes:

- A tool, described in [2] that hierarchically builds the behaviour model of a component system. At each level of the component hierarchy, it builds the automata describing life-cycle and binding behaviour.

  We are now working to add the Fractive new elements to this tool, namely the automata encoding the request queue, the proxies for future responses, the NewServe policy for primitives and the RunActive policy for composites as described in Section 3. This tool produces networks of parameterized automata in Parameterized FC2 format.
- A tool named Fc2Parameterized, described in [1], producing a finite instantiation of the system from a finite abstract domain for each parameter.

These values may be in some cases taken from the system description, as the buffer capacity set to 3 in the ADL, or deduced from the significant values occurring in the properties. For parameters which types are simple (see [1]) these abstractions are abstract interpretations in the sense of [8].

- A tool analysing the ADL. It generates the structure of the component hierarchy and the synchronisation networks for combining the various parts at each level of the system.
- Interface tools with the CADP tool-set [9], at the level of LTSs and of synchronisation networks. We then make a heavy use of the CADP tools (distributed state space generator, bisimulation minimiser, on-the-fly model-checker).

The length of Fractive request queues are unbound, and their abstraction must be chosen carefully. The choice of the queue depth is critical w.r.t. size of the generated state space: considering request queues of size 3, we were only able to generate the state space of **BufferSystem** (approx. 191M states and 1,498M transitions) on a cluster composed of 24 bi-processor nodes using the distributed model generation tool *distributor* from the CADP tool-set. For the complete system we did not even try to generate the complete automaton.

Staying in the context of explicit-state tools, we use a better approach : we define the set of control actions (whether in deployment or reconfigurations) involved in each specific property we want to prove. Then we forbid any other control actions for the model, and we also use this set to determine an approximation of the length of the queue. Given those parameters, we build all basic automata, hide any action not involved in the properties, and reduce the basic automata w.r.t. weak bisimulation. Last, we compute the products of the reduced automata, using the on-the-fly verification feature of CADP. This approach has enabled us to verify all properties listed in the next section in a simple desktop machine (CPU Pentium 3GHz, RAM 1.5 GB).

A potential gain would be to use partial orders or symmetry based state-space representation, especially for the request queue structures, and depending on the commutativity properties of the service policy.

5 Properties

The preceding sections focused on building the correct models and not on expressing properties. This section presents some properties to illustrate the verification power of our approach. We use regular alternation-free μ-calculus [10] because of its rich expressiveness and because it is the default way to express properties in the model-checker we use (from the CADP tool-set).

Regular alternation-free μ-calculus is an extension of the alternation-free fragment of the modal μ-calculus with action predicates and regular expressions over action sequences. It allows direct encodings of "pure" branching-time logics like CTL or ACTL, as well as of regular logics like PDL. Moreover,

it has an efficient model checking algorithm, linear in the size of the formula and the size of the LTS model.

5.1 Deployment

In Section 3 we defined the *deployment* automaton, that describes the control steps required for setting the system elements and bindings, and starting all components. For synchronous components [2], the *static* automaton represents the normal behaviour of the component after deployment.

In Fractive, however, method calls are asynchronous, and there may be delays between the request for a control method and its treatment. So checking the execution of a control operation must be based on the observation of its application on the component, rather than the arrival of the request:

- The actions `Sig(bind(intf1,intf2))` and `Sig(unbind(intf1,intf2))` encodes when a binding between the interfaces `intf1` and `intf2` is effective. It corresponds for instance to the synchronisations `!bind/unbind`(E_RI^{nr}, I_{ext}) or `!bind/unbind`$(I\_RI^{np}, SubC^k.E\_PI^{scnp})$ in Fig. 3.
- The actions `Sig(start(name))` and `Sig(stop(name))` encodes when the component `name` is effectively started/stopped. It corresponds to the synchronisations `!start/stop` in Fig. 3.

One of the interesting properties is that the hierarchical start operation effectively occurs during the deployment; i.e. that the component and all its sub-components are at some point started. This property can be expressed as the (inevitable) reachability of `Sig(start(name))` in the static automaton of `System`, for all the possibles executions, where `name` = {`System, BufferSystem, Alarm, Buffer, Consumer, Producer`}. We leave the actions `Sig(start(name))` observable in the static automaton and we express this reachability property as the following regular μ-calculus formula, verified in our example:

`[ true*.Sig(start(System))] true` $\wedge$ `[ true*.Sig(start(BufferSystem))] true` $\wedge$
`[ true*.Sig(start(Alarm))] true` $\wedge$ `[ true*.Sig(start(Buffer))] true` $\wedge$
`[ true*.Sig(start(Consumer))] true` $\wedge$ `[ true*.Sig(start(Producer))] true` **(1)**

5.2 Pure-Functional Properties

Most of the interesting properties concern the behaviour of the system after its deployment, at least while there are not reconfigurations. For instance, in the example, we would like to prove that a request for an element from the queue is eventually served, i.e. that the element is eventually obtained. If the action of requesting an element is labelled as `get_req()` and the answer to this request as `get_rep()`, then this inevitability property is expressed as the following μ-calculus formula, as well verified by the static automaton of the example:

`[ true*.get_req() ]` μ `X. (<` $\neg$ `true > true` $\wedge$ `[` $\neg$ `get_rep() ] X )` **(3)**

5.3 Functional Properties Under Reconfigurations

The approach described in this paper enables to verify properties not only after a correct deployment, but also after and during reconfigurations. For instance, property (3) becomes false if we stop the producer since at some point the buffer will be empty, and the consumer will be blocked waiting for an element. However, if the producer is restarted, the consumer will receive eventually an element and the property should become true again. In other words, we can check that, if the consumer requests an element, and then the producer is stopped, if eventually the producer is started again, the consumer will get the element requested.

For proving this kind of properties the static automaton is not sufficient, we need a behavioural model containing the required reconfiguration operations. We add to the component network a *reconfiguration controller* (Fig. 10): its start state corresponds to the deployment phase, and the next state corresponds to the rest of the life of the component, where reconfigurations operations are enabled but are no more synchronised with the deployment. This state change is fired by the successful termination of the deployment ($\surd$).

Fig. 10. Synchronisation product supporting further reconfigurations

For the property stated above, the reconfigurations `?stop(Producer)` and `?start(Producer)` are left visible, and this property is expressed by the μ-calculus formula, which is also insured in our example :

```
(* If a request from the consumer is done before reconfiguration *)
[ (¬ (?stop(Producer) ∨ ?start(Producer))*.get_req() ] (
   (* a response is given before stopping the producer *)
   μ X . (
      < ¬ ?stop(Producer) > true ∧ [¬ (get_rep() ∨ ?stop(Producer))] X)
∨
   (* or given after restart the producer and without stopping it again *)
   [ true* .  ?start(Producer) ] μ X . (
      <¬ ?stop(Producer)> true ∧ [¬ (get_rep() ∨ ?stop(Producer))] X))  (4)
```

5.4 Asynchronous Behaviour Properties

Let us now focus on a property specific to the asynchronous aspect of the component model. The communication mechanism in Fractive allows any future, once obtained, to be updated with the associated value, provided that the corresponding method is served and terminates correctly; binds, unbinds or stops operation cannot prevent this. For example, if the consumer is unbound after a request, it gets anyway the response, even if the link is then unbound or the component stopped. Using the approach for reconfigurations described

above: enabling `?unbind(buffer,Buffer.get)` and `?stop(Consumer)`, the property can be expressed as follows. This property is verified in the example:

$$\texttt{[true*.get\_req()] } \mu \texttt{ X. (< true > true } \wedge \texttt{ [} \neg \texttt{ get\_rep()] X)} \quad \textbf{(5)}$$

6 Conclusion

This paper provides methods and tools to build the specification of distributed hierarchical components, in a hierarchical bottom-up fashion. Our approach relies on the definition of a synchronisation network of LTSs, each LTS expressing a different aspect of the component behaviour. The functional behaviour of primitive components are given by the user either with an specification language or obtained by data source analysis. The non-functional behaviours are automatically incorporated based on the component description. Then a set of properties is described and proved on a component system example. Some of those properties, e.g. dealing with deployment, concern any component system and can be verified in a systematic way.

The main contributions of this paper are:

- We define a general synchronisation network modelling the functional and control behaviour at any hierarchical level of a component system.
- We adapt [1] for active objects and model the behaviour of active (primitive and composite) components.
- In both primitives and composites we focus on the Fractive implementation of distributed components. We incorporate the Fractive component features by automatically adding automata encoding the queues, future responses and serving policies depending on the life-cycle status
- Finally we prove a set of properties classified upon the component life phase, and considering the asynchronous aspects of Fractive components.

The model is automatically built from the functional behaviour of primitives and the component system description (as a XML file). We have illustrated our approach with a guided example: we described step by step the automatic construction of the model, and we discussed techniques to avoid the state explosion problem.

Finally, many approaches are being developed to cover the right composition of components considering their functional aspects. From the user point of view, one of the strongest advantage of components is the separation of concerns. However, when coming to behavioural verification, one still needs to take into account the inter-play between functional and non-functional aspects, at least for existing component models. The main originality of this paper is to encode the deployment and reconfigurations as part of the behaviour, and thus verify the behaviour of the whole system of components.

This paper provides a big step towards a concrete and strongly usable tool-set. This tool-set builds the models automatically and gives feedback about

generic properties and errors detection. The user may also define and verify further properties, or use the generated models to check against a specification.

Further developments are necessary to take into account other important features of distributed systems, in particular the managment of exceptions, and the mechanisms for group communication, very important in computing Grids. We also want to adopt more efficient techniques for the analysis and model-checking tools, improving our use of on-the-fly methods, and looking at specific, more compact representation of state-spaces.

References

[1] T. Barros, R. Boulifa, and E. Madelaine. Parameterized models for distributed java objects. In *Forte'04 conference*, Madrid, 2004. LNCS 3235, Spinger Verlag.

[2] T. Barros, L. Henrio, and E. Madelaine. Behavioural models for hierarchical components. In *SPIN'05 Workshop*, San Francisco. LNCS 3639, Spinger Verlag.

[3] F. Baude, D. Caromel, and M. Morel. From distributed objects to hierarchical grid components. In *International Symposium on Distributed Objects and Applications (DOA)*, Catania, Italy, 2003. LNCS, Springer.

[4] E. Bruneton, T. Coupaye, and J. Stefani. Recursive and dynamic software composition with sharing. 7th ECOOP Int. Workshop on Component-Oriented Programming (WCOP'02), June 2002.

[5] A. Fantechi C. Carrez and E. Najm. Behavioural contracts for a sound assembly of components. In *in proceedings of FORTE'03*, volume LNCS 2767. Springer-Verlag, 2003.

[6] D. Caromel, L. Henrio, and B. Serpette. Asynchronous and deterministic objects. In *31st ACM Symp. on Principles of Programming Languages*. ACM Press, 2004.

[7] D. Caromel, W. Klauser, and J. Vayssière. Towards seamless computing and metacomputing in Java. *Concurrency Practice and Experience*, 10(11–13):1043–1061, Nov. 1998.

[8] R. Cleaveland and J. Riely. Testing-based abstractions for value-passing systems. In *CONCUR'94*. LNCS 836, Springer, 1994.

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

[10] R. Mateescu and M. Sighireanu. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. In S. Gnesi et al, editor, *Proceedings of FMICS'2000*, GMD Report 91, pages 65–86, Berlin, April 2000.

[11] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11), nov 2002.