Specifying Fractal and GCM Components With UML

Solange Ahumada*, Ludovic Apvrille†, Tomás Barros‡, Antonio Cansado§, Eric Madelaine§ and Emil Salageanu§
*Univ. Técnica Federico Santa María, Valparaiso, Chile, Email: sahumada@elo.utfsm.cl
†GET/ENST/LabSoC, Institut Eurecom BP 193, 06904 Sophia-Antipolis Cedex, France, Email: Ludovic.Apvrille@enst.fr
‡Univ. Diego Portales. Ejército 441, Santiago, Chile, Email: tomas.barros@udp.cl
§ INRIA Sophia-Antipolis, CNRS/I3S/UNSA. BP 93, 06902 Sophia-Antipolis Cedex, France. Email: First.Last@sophia.inria.fr

***Abstract*—UML 2 has introduced new diagrams for expressing hierarchical structures and their assembly, and has brought some new features to the behaviour-oriented diagrams (activities and state machines), that help modelling component systems. However, UML leaves many semantic decisions opened, and various emerging component frameworks also have features that cannot be directly expressed using UML 2 concepts. In this paper we present an approach for modelling two different component frameworks using UML 2 diagrams. First we define a mapping between the Fractal component model and UML 2 diagrams, and we describe CTTool, that allows to edit and model-check diagrams for Fractal components. Then we propose an extension of this work for the Grid Component Model, that is an extension of Fractal providing asynchronous, collective, and autonomic features for distributed component systems.**

***Index Terms*—software requirements engineering, software components, software reliability, grid computing**

This work is supported by the french ACI securité FIACRE, by the EC FP6 Streps project GridCOMP, and by the SticAm-sud collaborative project Reseco.

I. Introduction

In software engineering, a strong emphasis on system specification methods and tools has been made. Among several techniques, Component-Based Software Development (CBSD) has emerged as a response for decoupling designs into partially independent and reusable modules. It has been adopted by UML 2 [1] through Component Diagrams, and it provides a semi-formal way of specifying components. CBSD's main idea is to explicitly specify all possible interactions with the environment through well defined entry points while hiding all internal details.

Informal specifications allow an easy and fast development and are useful for non-expert users, but the lack of formality leads to ambiguity. On the other side, formal specifications require user expertise and usually lead to a longer specification time, but provide a precise syntax and semantics which suits computer-aided verification such as theorem provers and model-checkers.

Further, specifications may also be textual or graphical. Textual specifications allow enough granularity for specifying details in the requirements, whereas graphical ones bend better to coarse system specifications. Moreover, graphical specifications are more intuitive and require shorter learning and shorter development time than textual ones.

We state that CBSD fits well with both graphical languages and formal languages. Then, we propose a new language based on UML 2 and endowed with a formal semantics. This language is precise enough to allow verification and model-checking of systems. The goal is not to go all through complex formal languages, but to take into account software engineers expertise of both syntax and semantics.

Amongst state-of-the-art component models, we are particularly interested in Fractal [2] and in the Grid Component Model (GCM) [3]. Fractal is a hierarchical component model including features for non-functional management and (re)configuration, and provisions for the specification of behaviour protocol between components. The Grid Component Model is an extension of Fractal dedicated to distributed applications, with specificities like deployment description, multicast/gathercast communications, and autonomic components. There exists a graphical editor for Fractal architecture descriptions, but no modelling tool that would include both architecture and behaviour specifications, and that would allow for a more or less automated analysis of the behaviour of component systems. Our work is a step toward this goal, using a mapping of Fractal/GCM component frameworks into UML diagrams.

Related work have been published, mainly in the domain of embedded or real-time systems, or in some more general distributed system context. Contributions related to distributed systems generally focuses on the modelling of protocols involved in underlying architectures [4]. Thus, protocol modelling has been addressed in UML since its very first versions, at first, with academic case studies [5], and then, with industrial-based large-scale applications in e-commerce for example [6].

More precisely, one of the contributions for modelling distributed applications is called AUML [7]. It relies on the normalized version of UML. The notion of *Protocol diagrams* is described: UML sequence diagrams are extended with the notion of logical operators to explicitly model causality, synchronisations and broadcast. Another contribution relies on UML *Interaction diagrams* to model interconnection between components. Those two contributions address the modelling of components in UML 1.4 and therefore ignore composite structure diagrams introduced in UML 2.

A more recent contribution extends UML with a notion of components, with in and out ports, and behaviour diagrams to describe critical applications [8]. However, their behaviour diagram notation is quite specific (using basic logical operators such as xor, not, etc.) and do not support complex protocols and buffer management as found in distributed systems.

In the area of embedded and real-time system, Omega [9] has been one of the first models of components, based on a subset from UML 1.4. It is endowed with a formal semantics based on statecharts.

The Architecture Analysis and Design Language (AADL) [10] is an architecture description language for component-based specification of real-time, embedded, software-intensive systems and for automated component-based system integration. It has a full UML 2 profile (and a MARTE profile), source of several tool environment, including an integration in the Topcased platform.

AEmilia [11] is an Architecture Description Language based on Stochastic Process Algebras. With AEmilia, the description of a system can be built compositionally and hierarchically through a graphical support. In addition to behavioural and functional verification, it is strongly targeted at performance modelling and analysis, and has been recently subject to UML tool development.

TURTLE [12] is another related work, initially not targeted at component systems. It is a UML profile dedicated to the modelling and formal verification of real-time systems. One of the strengths of this profile is its formal semantics. Indeed, all TURTLE diagrams are first translated into an intermediate form called TIF - TURTLE Intermediate Format - from which formal specifications into LOTOS-based process algebra can be generated. Moreover, the TURTLE profile is supported by a (open source) toolkit named TTool, developed by the LabSoC laboratory from GET/ENST, including editors for various UML views of the systems, and code generators for interfacing with LOTOS and LOTOS-RT model-checking tools. It appeared that the toolkit was a very convenient basis for developing our Fractal modelling prototype tool.

Our contribution in this paper is:

- A UML-based framework and tool for specifying and model-checking software components,
- A novel UML profile proposal, refining the previous framework, dedicated to distributed and asynchronous software components, with a stress on grid applications.

This paper is organised as follows: Section II gives some background on the Fractal and GCM component models, and on the component-related concepts in UML 2.x. In Section III we present a method for specifying Fractal components in UML, giving a specific interpretation to component and state machine diagrams, and show how CTTool implements this method. Then in Section IV we propose an extension of those models to deal with asynchronous distributed components. Finally Section V concludes.

II. Background

A. UML 2

UML [1] is the most widely used modelling language. In its latest version, component architectures may be specified using *Component diagrams*. As for all UML concepts, component diagrams are defined in a high-level and underspecified semantics (with semantics variation points), in order to deal with several existing component models (e.g. EJB, CCM, COM+ and .NET). Also, UML 2 doesn't provide any methodology for using UML components, and adapting them to more concrete models.

UML component diagrams feature both black box and white box views of hierarchical component systems. UML components communicate and synchronise through well-defined (provided and required) interfaces and connectors. The white-box view is used to specify the implementation of a component, in term of subcomponents and bindings, or in term of other realisation artifacts. In the black-box view, UML specifies that a protocol state machine can be attached to each component. This state machine defines the acceptable external behaviours of the component, and can be used to check the behavioural correctness of component assemblies.

Other usual UML diagrams are used to define the signature of component interfaces (with their operations and events) or the classes for data structures.

B. Fractal

Fractal is a hierarchical component model with a few but well defined concepts such as component, controller, content, interface, and binding. A Fractal component is formed out of two parts: a *controller* and a *content*. The content of a component is composed of (a finite number of) other components, called *subcomponents*, which are under the control of the controller. If a component does not have subcomponents it is called a *primitive* component, otherwise it is a *composite* component. This allows for hierarchical composition, in the sense that components may be nested at any arbitrary level.

The controller of a component can have *external* and *internal* interfaces. A component can interact with its environment through *operations* at its external interfaces, while internal interfaces are accessible only from its sub-components.

Interfaces can be of two sorts: *client* and *server*. A server interface can receive method invocations while a client interface emits methods call. A *functional* interface provides or requires functionalities of a component, while a *control* interface is a server interface that corresponds to a "non functional aspect", such as introspection, lifecycle, configuration or reconfiguration.

C. GCM and ProActive

The Grid Component Model (GCM) is a novel component model being defined by the european Network of Excellence CoreGrid, as an extension of Fractal.

Grids consider thousands of computers all over the world, and address heterogeneous architectures (from multi-core processors, to P2P local area networks, and to clusters and massively parallel machines). For dealing with latency in such networks, and enhancing the overall efficiency, the GCM uses asynchronous method calls. Grid applications also have numerous similar components, so GCM defines collective interfaces, providing dedicated synchronisation and distribution policies, and easing design and implementation of such systems.

A GCM reference implementation is being realised with ProActive [13]. In this implementation, one active object is assigned for each primitive component and for each composite membrane. As a consequence, this implementation also inherits some properties and constraints w.r.t. the programming model:

- components communicate through asynchronous method calls with transparent futures. These futures are first order objects: they can be forwarded to any component in a non-blocking manner;
- there is no shared memory between components;
- a single control thread is available for each component, either primitive or composite.

Method calls use a rendez-vous protocol: requests are enqueued in the callee (server) side in an atomic way. During the rendez-vous, a future is created as a placeholder for the returned result, whenever it arrives. Caller execution may freely continue up to a point where the concrete value of the result is needed. At this moment it is blocked until the concrete value is available; this implicit synchronisation mechanism is called wait-by-necessity. A precise operational semantics of ProActive is given by the ASP-calculus [14]; it allows to prove generic and important properties of ProActive's constructs on top of which we base our graphical language shown in Section IV.

The rest of the paper presents our work in two steps. We first consider Fractal components, and explain the mapping of its main concepts (except non-functional interfaces) to UML 2 component diagrams and state machine diagrams and its implementation in the CTTool software. This will be of interest for many of the developments and implementations done in the Fractal community. The following part is specific to the GCM model, or more generally to distributed component systems using remote method invocation as their basic communication mechanism.

III. Specifying Fractal Components with UML

There already exists work for modelling hierarchical components with UML. One of the closest to our goals is from Mencl and Polak [15], who discussed various possible mappings between the Fractal and UML 2 key concepts, though they only address the architectural part of Fractal, not the behaviour description. Our architectural diagrams are very close to theirs, and we add the behaviour description in the form of (protocol) state machine diagrams.

We have implemented these diagrams in a tool called CTTool. It is based on TTool and we give its syntax and semantics below.

For the drawings in this paper we use the classic Producer-Consumer example (see Fig. 1). This example is composed of:

- a bounded *Buffer* modelled as a composite with two subcomponents: *BufferQueue* where method calls are stored until they are served by the *BufferBody*; and *BufferBody* expressing its functional behaviour (serving from the queue and processing the request);
- one *Producer*, modelled as a primitive component. Elements are sent one at a time to the buffer by calling a method named `put` on the *Buffer*;
- and two *Consumers* each modelled as a composite with two subcomponents: *ConsumerBody* with its functional behaviour (consuming elements through a method call `get`); and *ConsumerProxy* which takes care of the remote method call – it performs the call and receives the return value.

A. Architectural Modelling

UML 2 provides Component Diagrams for specifying the architecture of components. In Fractal this is done with the Fractal Architecture Description Language (ADL). So, for structural modelling of a Fractal component we map the Fractal ADL to UML 2 Component Diagrams as follows:

1) Components and Subcomponents: The two Fractal component kinds, *composites* and *primitives*, can be distinguished according to the existence of subcomponents. In composites there are subcomponents but not in primitives. Then, Fractal ADL components can be designed using a UML 2 metaclass `Component`, and subcomponents can be defined using an embedded definition of a subcomponent within a definition of its owning component.

When modelling a system with a compositional approach, there is no need to specify whether a given component is primitive. This allows components to be further refined if needed, or to give a final specification of them.

2) Provided and Required Interfaces: UML offers a more complex interface structure than Fractal, as it has both *ports* (that can be unidirectional or bidirectional), and *interfaces* (that are either provided or required); a port may contain several interfaces. In order to encode in a compact way Fractal interfaces and to obey the semantics of the remote method invocation mechanism, we define in CTTool a notion of oriented `port`. A CTTool `in-port` (resp `out-port`) models a Fractal provided (resp required) interface, and is composed of:

- a UML *port*, which name is the name of the Fractal interface,
- exactly one provided (resp required) *interface* on which the component receives (resp emits) the service requests,
- exactly one required (resp provided) *interface* on which the component sends (resp receives) back the request result,
- a boolean attribute stating whether this port is mandatory or optional (the interface contingency in Fractal vocabulary).

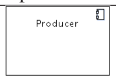
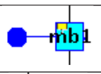
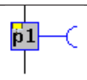
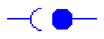
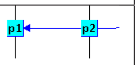
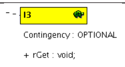
CTTool Element	Representation
Component – composite or primitive Fractal component	
IN port – represents a Fractal server interface	
OUT port – represents a Fractal client interface	
Connector – represents a Fractal binding	
Delegate connector – represents a Fractal binding between a subcomponent interface and its parent inner interface	
Interface signature – defines the interface's methods	

TABLE I
ELEMENTS OF A COMPONENT DIAGRAM

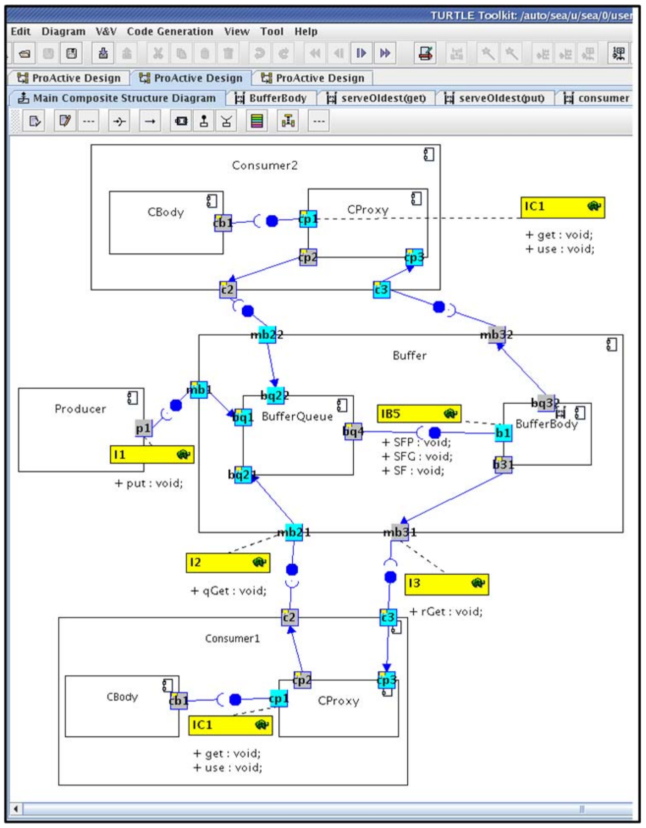


Fig. 1. Component Diagram for the Producer-Consumer architecture.

3) Bindings: Fractal ADL provides both *primitive* and *composite* bindings. The later allow for building various complex communication schemes, but we do not need them, at least in the current state of our framework. CTTool only implements *primitive bindings*, using UML *assembly* and *delegate* connectors. Connections may be between `in` and `out` ports of components having the same parents or between a component and its own subcomponent and vice versa. Naturally, connected ports must have compatible signatures.

Table III-A1 shows a graphical representation of the elements explained before.

The full component diagram of the Producer-Consumer, built using CTTool, can be seen in Fig. 1. Some of the signatures were hidden for simplicity; those that are shown display their list of methods, with their return type (a *void* return type implies that there is no associated result message). Note that some of the primitive components in this diagram have an additional icon, that indicates they have a state machine diagram associated, specifying their behaviour. Remark also that we have included two instances of a Consumer component, this is because Fractal does not allow for multiple components in its architecture description; we shall discuss this point later in the paper.

B. Behavioural Modelling

For the behavioural modelling we chose to use UML 2 State Machine Diagrams. Indeed, these diagrams specify sequences of events managed by various components. In our case, incoming events are reception of incoming method calls, and of remote method call results; outgoing events are emission of remote method calls, and of incoming method calls results.

Components are unaware of the structure of their environment, except for events occurring at their interfaces. A black-box view of the component represents the protocol constraining these interactions with the environment. This is done by associating a state machine exposing the component's functionality. For every primitive component (meaning here a component with no content), it is mandatory to give a State Machine Diagram with its behaviour, whereas for composites it is optional when the component is provided with a respective Component Diagram. This constraint ensures that the global behaviour of the system can be generated by CTTool.

We consider method calls to be composed of two separate messages: the call itself, and its return value (for non-void calls). Method calls may have arguments with a value passing CCS-like semantic, but in the current CTTool prototype the only allowed type is integer – we plan to support real Java user-classes as discussed in Section IV-B. These messages are of the form: `<messageName>!<expr>` for sending requests, and `<messageName>?<var>` for receiving back results. In these forms, `<expr>` can be a variable or a set of operations involving variables; it is mandatory to have previously declared all variables. Variables are visible (and shared) by all submachines a machine may have.

Table II shows a graphical representation of the state machine operators: State machines diagrams allow developers to define states, to send or receive messages over ports, to synthesise behaviour with submachines (submachine operator), to define complex transitions between states (with choice

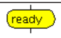
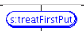
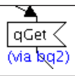
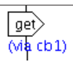
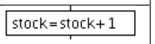
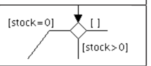
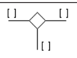
Operator	Representation
States	
(named) State	
Start point	
Stop point	
Submachine	
Transitions	
Receive message	
Send message	
Action	
Choice	
Non-deterministic choice	

TABLE II
OPERATORS OF A STATE MACHINE DIAGRAM

operators, guards, and mixing of data and control flow), to specify actions over variables, and to abstract away from implementation choices (non-deterministic choice operator). Moreover, there are distinguished start and stop operators for defining initial and final machine states. The stop operator returns execution to its parent machine; if the machine is a root, the machine halts.

When a method call is performed over a bound interface, two components synchronise on the message call, and some time later on the return value message; it means that the call and the response are not atomic. If a method call is performed over an unbounded interface, the component halts.

In Fig. 2 the state machine diagram of the *BufferBody* component is shown as an example. In this case, the use of submachines in the state machine diagram is the developer's choice. The *choice operator* is used with guards controlling the buffer state; when `empty` only a `put` method is allowed; when `full` only a `get` method is allowed; otherwise both are allowed.

C. Practical Experience

One can wonder to which extent this kind of approach may scale up, in term of expressivity and convenience of the graphical language, and in term of feasibility of model-generation and model-checking.

We have modelled with CTTool a full-fledged case-study called Common Component Modelling Example (CoCoME [16]) using the techniques and tools shown above. As its name suggests, CoCoME is an initiative for defining a common component example, and may be used for comparing different component models. The subset of CoCoME we modelled consists in 16 components, 5 of them being composites. Furthermore, composite components were designed with up to 5 layers of hierarchy, stressing the need of hierarchical

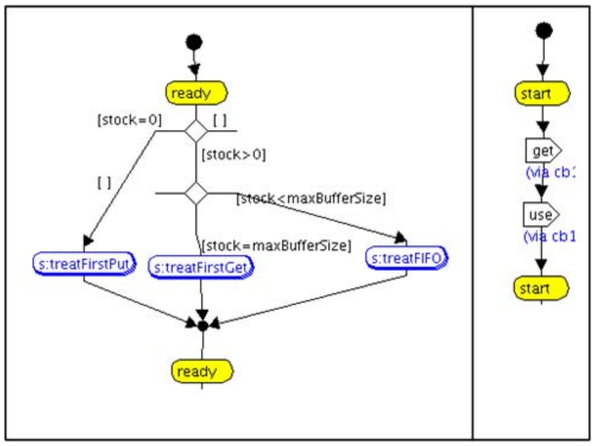


Fig. 2. State Machine Diagrams for *BufferBody* and *CBody* in CTTool.

component models. In fact this system necessitates hierarchical components, component multiplicities, collective communications (for addressing a specific component and broadcasting a message), modelling of exceptions, and synchronous and asynchronous method calls.

CTTool generates formal code in LOTOS for either the global system behaviour, a single component, or an arbitrary assembly. Moreover, it has a bridge towards the CADP toolset [17], which in principle allow:

- simple "press-button" properties, like the absence of deadlocks, or the absence of certain types of events (predefined error events);
- more complex temporal properties expressed as temporal logic formula, or in a formalism that can be translated into the temporal logic language understood by the model-checker;
- conformance between the implementation of a component (computed from the behaviour of its structure) and its black-box specification, expressed as an equivalence or a preorder relation (as in [18]).

When generating a LOTOS model for model-checking in CADP, the CTTool user must give finite instantiations of its data parameters, depending on the formula he/she wants to check. We specified instantiations in such a way that we could check 6 formulas (expressing various usage scenarios) defined by CoCoME. The generated model had 81 distinct transition labels (instances of communication events). Its size before reduction was 1.25 million states / 3 million transitions, and after reduction by branching bisimulation 9800 states / 33 000 transitions.

As a result of model-checking this model, we were able to find several errors within the reference specification. Most of them were related to underspecification of requirements, others to non-trivial race conditions within the system. The results of this experiment are available in [19].

D. Evaluation of the current framework

At the current point of our framework development, we have defined a mapping of Fractal architectural and behavioural features into UML 2 diagrams (based on Mencl and Polak work for the architectural part), and implemented a prototype based on the TTool toolkit that enables us to perform generation of finite-models and model-checking of temporal logic formulas for those diagrams. We do not address all concepts of the Fractal specification (in particular no composite bindings, no shared sub-components and no collective interfaces).

Within these limits, we have designed a full set of tools allowing the modelling of components, generation of finite, hierarchical, LTS-based representations of their behaviours, and model-checking of temporal-logic properties.

Technically we have used the internal TURTLE format of the toolkit, extended by a component hierarchical structure, to give the semantics of our diagrams. This can be viewed as an informal semantics (defined by the code), while the formal semantics is still to be provided. We intend to do this in term of the pNets model [20], that is a very expressive parameterized and hierarchical model for synchronised labelled transition systems.

IV. Specifying GCM/ProActive Components

In the previous section, we tried to be as compliant as possible with the Fractal specification. Most Fractal implementations can be modelled with the techniques described above. However, our specific goal is to address GCM/ProActive specifications (see section II-C), and there are several limitations in the current model that prevent the specification from capturing the desired behaviour. We also want to provide the GCM/Proactive developers with primitives allowing them to express their designs at the right level of abstraction. Here are the most important missing features, using our Producer-Consumer example:

- *Asynchronous Method Calls*: the GCM/ProActive model uses a queue in the server side, and proxies in the client side, whereas in CTTool these had to be explicitly modelled by the user by adding two additional components: *BufferQueue* and *ConsumerProxy*;
- *Service policy*: it is possible to define the activity of a component in GCM/ProActive but this is not captured by CTTool; we had to encode its behaviour description within a state-machine as in Fig. 2;
- *Multiplicity*: instantiation and binding of multiple instances of a component do not exist in CTTool, so multiple producers/consumers had to be explicitly instantiated at design level.

Additionally, CTTool does not offer constructs for expressing other GCM/ProActive features such as *Multicast and Gathercast Interfaces*, that are extensively used in Grid applications to develop applications that can take advantage of complex and massively distributed grid insfrastructures, without fixing the numbers of components in the source code. Additionaly, multicast and gathercast interfaces allow the specification of specific communication mechanisms, to optimise network usage, and to hide latency as much as possible.

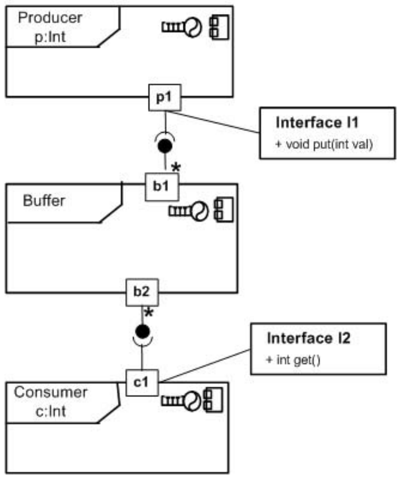


Fig. 3. *Active Components* representing GCM/ProActive component.

In the rest of the paper we propose answers to these constraints by extending the language with abstractions for the various aspects of *Active Components* listed above. For the time being we do not address the specification of Fractal/GCM non-functional management interfaces as well as of reconfiguration of dynamic architectures. We hope to include these features in the mid-term in order to answer a wider set of GCM/ProActive designs.

A. Language Extensions

We start by specialising the component diagrams for active components: a GCM/ProActive component provides a request queue that collects in an asynchronous manner the messages arriving on its provided interfaces, and a service thread that serves requests in the queue. Within the Component Diagram, this will appear as a novel kind of component with a dedicated icon nearby the component icon (see Fig. 3).

To ensure compact and flexible specification of Grid applications that may be deployed on large infrastructures, and where some subcomponents may represent groups of similar component instances, sharing some part of the computation, we provide a notation for parameterized active components. More precisely, an active component can have one or more parameters, that are indexes ranging over some built-in or user-defined type. A consequence of this is that we need a more expressive syntax and semantics for ports and connectors; indeed, both `in` and `out` ports of active components can be multiple when they are connected to parameterized components. The precise addressing or multicasting strategy will be defined in the associated state machine. In Fig. 2 we had two explicit instances of the Consumer component, with as many bindings from the Buffer to the components; in Fig. 3 we represent any possible configuration of a buffer with c Consumers and p Producers in a single diagram.

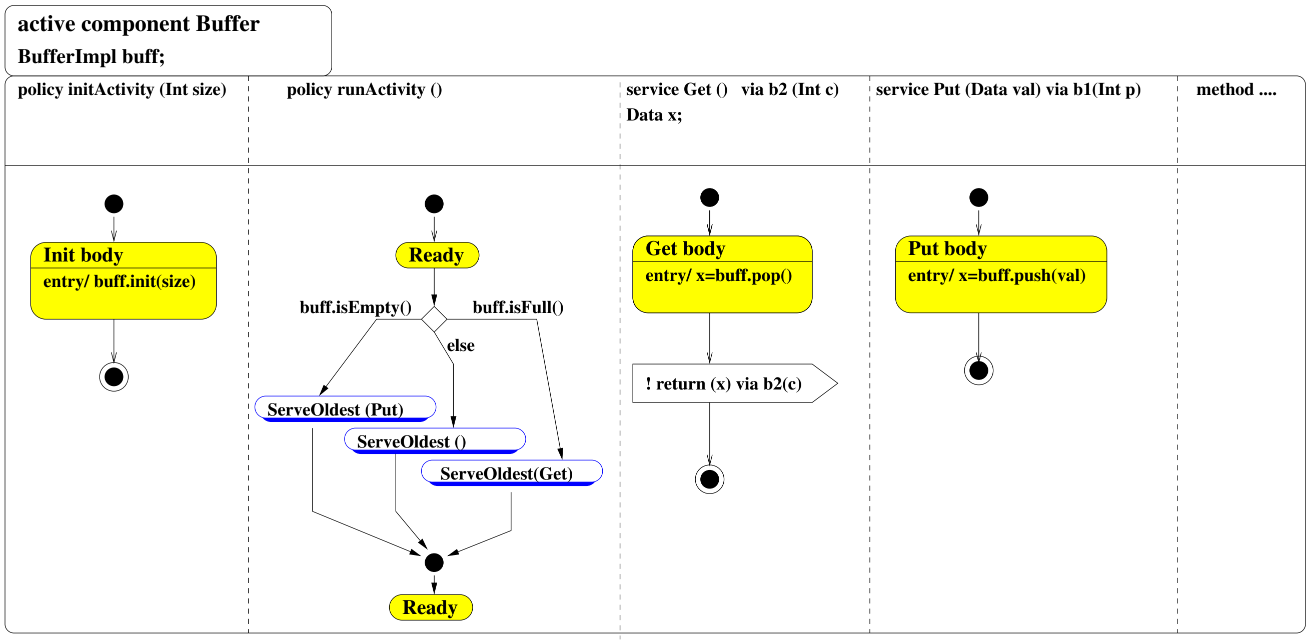


Fig. 4. State machines for the Buffer component

Then, queues and proxies no longer have to be explicitly defined by state machines as was the case in CTTool (compare Fig. 3 with Fig. 2); they will be handled implicitly by the diagram's semantics when generating the semantic representation of the components (see Section IV-B). The mechanism that links together those queues and proxies with the user-defined service policies is also implicit. Concretely, we introduce a new graphical construct for modelling the behaviour of an active component. This construct resembles a "region" diagram, but its header label starts with a "active component" keyword, and includes the component name and declarations of its instance variables. Its sub-regions contain the state-machines required for defining the service policy of the component; this includes the policies and the services state-machine, that we describe in the next section, but also auxiliary sub-machines that can be used to enhance readability of the diagrams.

a) Policies: The service policy is FIFO by default, or can be specialised by the application developer with a set of methods manipulating the queue. The component goes through three states in its lifecycle: `InitActivity`, `RunActivity`, and `EndActivity`. The first and last take care of initialisation and termination policies respectively, whereas `RunActivity` gives the component activity, usually as an infinite loop. An example of a `RunActivity` state-machine is shown in Fig. 4; the submachines in it are using predefined service policy methods from the ProActive API, named `Serve*`. These methods allow for various policies of selection and execution of requests in the queue, and that take as argument zero, one or more method names; e.g. `"ServeOldest(Put,Get)"` means "pick the oldest request in the queue matching one of the method names Put or Get.

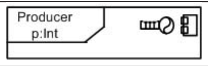
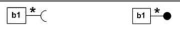
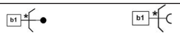
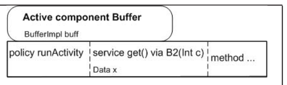
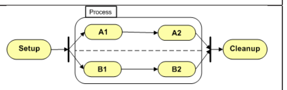
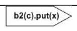
Graphical Element	Representation
Components	
(Parameterized) Active Component	
Multiple interfaces	
Multicast and Gathercast	
State Machines	
Active Component Behaviour (with local variables)	
Regions, Forks and Joins	
Request on a required interface	
Wait for a Future Update	

TABLE III
NEW GRAPHICAL ELEMENTS FOR ACTIVE COMPONENTS

b) Services: Then the behaviours of those methods (that are the public services offered by the component) have to be specified themselves by state machines. There must be one such state machine for each public method offered on a provided interface of the component, and it appears in the active component behaviour with a special header starting with the keyword `service`. This header has a specific syntax, taking care of: the bindings of the request arguments, but also, for services coming on multiple interfaces; and the binding of the parameters of the interface (after the `via` keyword). The detailed code of these services need not be detailed here (this will only be required in the implementation code later), but all communication events with other components (method calls on required interfaces) should be there, and also the part of the control flow and of the data flow that will influence the behaviour. Note in Fig. 4 that these machines may access and modify the local variables of the component, that were declared in the header label of the active component behaviour diagram, and shared between the services.

c) Component Synchronisation: How does this all work together? The formal model for these components was described in [21]. The various state machines encoding predefined or user-defined policies, the component provided services, and the automatically generated queues and proxies are assembled using a synchronisation network, that is also automatically deduced from the component diagram. This network implements the GCM/ProActive semantic model, as described in Section II-C.

Examples: the `RunActivity` state machine of the Consumer component is a simple loop that repetitively calls the `get` method on its `C1` interface, then waits (explicitly) till it gets an answer. The `get` call appears within a *send message* transition event, that corresponds (implicitly) to the creation of a future proxy. The next transition event is a special synchronisation action `use` that is forcing the component thread to wait for the return of the `get` result.

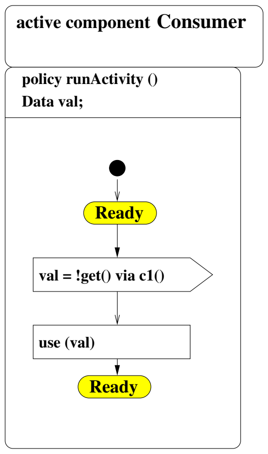


Fig. 5. State machine for the Consumer component

d) Distributed or Concurrent Services: Although GCM/ProActive components have a unique service thread, it is necessary to define concurrent services within the component activity. A composite component delegates services to its subcomponents, each of them having its own service thread. But we do not want designers to give implementation details of the component architecture within state machines. So, in general, the black-box specification of a component may have one or more parallel activities, defined as multiple services within the `RunActivity`. This is done using fork operators, denoting services running in parallel. Each one of these services defines its own service policy and its own local set of variables.

e) Interfaces between multiple components: Then, collective interfaces may be defined for dealing with distribution and synchronisation of method calls amongst groups of distributed components. We propose graphical primitives for representing multicast and gathercast interfaces (Fig. 6). These were inspired by [22], and are:

- *Multicast client interface*: a client interface connected to several (N) server interfaces. The designer may specify the distribution policy of the method calls and the method's arguments. Among those policies we distinguish broadcast and scatter; broadcast sends the same message, whereas scatter splits the method's arguments into N pieces and distributes these towards the servers.
- *Gathercast server interface*. Symmetric to a multicast interface, N client interfaces are connected to a single server one. Multiple messages are gathered, and a single call is sent to the server component. This interface also works as an implicit barrier, synchronising all clients. The designer may specify the distribution policy of the method's results symmetrically to the multicast.

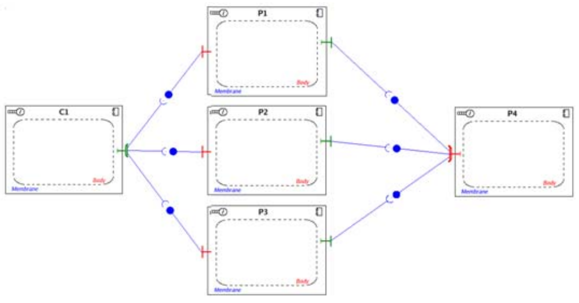


Fig. 6. Multicast interface on the left, and Gathercast interface on the right

B. Model Generation

For the user-defined part of the component's behaviour, and for their synchronisation, the model generation is the same than in the current CTTool implementation. In addition, modelling of GCM/ProActive's queues and proxies will have to be automatically generated during the translation into the internal model, before translation to the model-checker format.

A queue is created for each *Active Components*. For that, we use the model defined in [21] of a ProActive component and create automata encoding its behaviour. Note that as we currently work with finite-state model-checkers, an instantiation of the queue is mandatory. An example of such queue with length of 2 is given in Fig. 7 expressed as a state machine in CTTool. Roughly, there is a state for each combination of pending methods within the queue, and transitions over incoming and outgoing events, i.e. push/pop. Although the diagram looks complex, it is not visible by the user.

Finally, for each future within state machines, we create a proxy. The automaton takes care of the remote call and of the

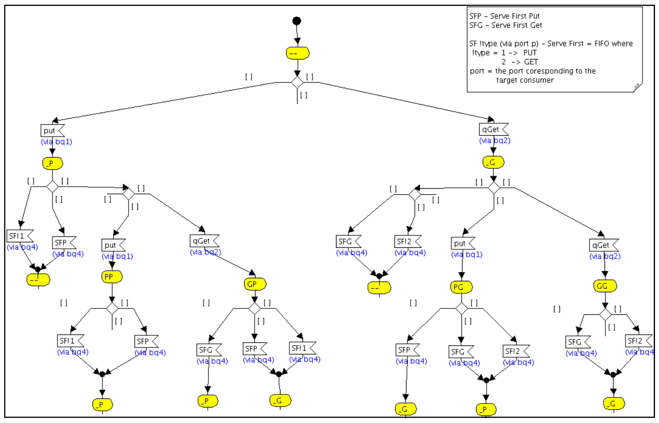


Fig. 7. Finite instantiation of the buffer queue

future value update. Within CTTool, a similar behaviour can be achieved with the state machine shown in Fig. 8; similarly as above, the tool handles these proxies automatically.

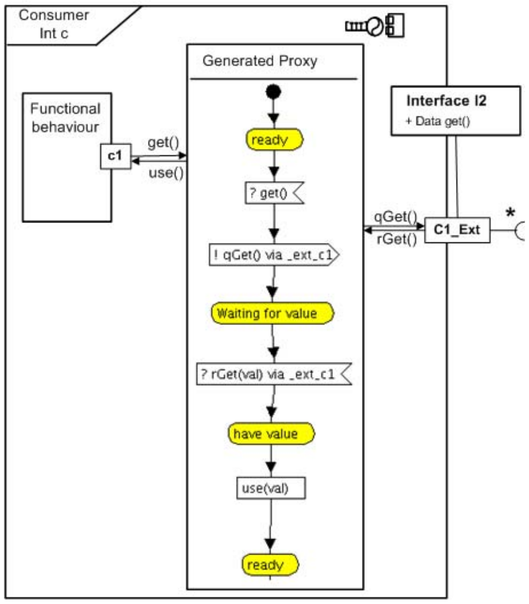


Fig. 8. Generated proxy for the Consumer component

Note that some static analysis of the component state-machine can be used to determine the points in the graph where the future values are used. There are cases where it can be automatically proved that a future value is used exactly once in each possible execution path; as a consequence the corresponding proxy machine can be terminated at this point (this is the case in Fig. 8). A difficulty is that GCM/ProActive allows futures to be passed as arguments to remote components (this is called automatic continuation); so in absence of information on the context, potentialy all arguments of received requests can be futures, or be structures containing futures. This feature makes the analysis more difficult, and is subject of further research.

V. Conclusion

In this paper we have shown how Fractal and Grid Component Model (GCM) components can be specified using UML 2 diagrams for both architectural and behavioural specifications. For the architectural specification we based our work on previou work by Mencl and Polak [15], leaving our main contribution on the behavioural specification. For the latter, we used UML's State Machine Diagrams endowed with formal semantics denoting precisely the behaviour of Fractal components.

The graphical specification language is formal enough to be model-checked, and it fits well with most software engineer's expertise as it is based on widely used UML 2 diagrams. Further, the language allows for incremental refinement of components, meaning that at each level a component specification may be refined if needed, or a final implementation may be specified. We have developed this framework using only existing UML diagrams, and yet it is applicable to various implementations of the Fractal specification. Indeed, we support this language in a tool called CTTool, and we tested our methodology by specifying a large scale case-study. We have both the tool and the case-study available at our website [1].

In a second phase, we also give the basis for addressing distributed components specification, concretely in the domain of GCM/ProActive components. For that, we propose to create a new UML profile which brings constructs and semantics for dealing with *distributed active components*. An advantage of this approach is that we let designers capture the business behaviour of the system, not the complex semantical encodings of the underlying communication protocol. Moreover, the language is extended to deal with parameterized (multiple) components and collective interfaces. These fit well with system designs taking into account large number of nodes distributed on the Grid, and allow precise and concise specification of distributed components.

CTTool is used to create formal LOTOS specifications that suit as input for state-of-the-art model-checkers. However, the diagnostic relies on the generated model, not on the user model. Therefore, it may be complex to understand the nature of the problem. The solution is to find a safe mapping back to the specification. We expect to study further this issue in the next release of our tool.

Finally, we also plan to study the conformance of a component implementation given by a component diagram (and the set of subcomponents' state machines) with its black-box behaviour given by a state-machine. Ior the moment we leave unspecified the meaning of the equivalence (or preorder). Many existing work can apply here, starting with all notions of simulations and bisimulations inherited from process algebras. They have to be adapted to our component model though, e.g. in a way similar to the component substitutability relations of [18].

[1]http://www-sop.inria.fr/oasis/Vercors

REFERENCES

[1] *UML 2.0 Superstructure Specification, http://www.omg.org/cgi-bin/doc?ptc/2004-10-02*, omg, Oct. 2004.

[2] E. Bruneton, T. Coupaye, M. Leclercp, V. Quema, and J. Stefani, "An open component model and its support in java." in *7th Int. Symp. on Component-Based Software Engineering (CBSE-7)*, ser. LNCS 3054, may 2004.

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

[4] H. Gomaa, *Designing Concurrent, Distributed, and Real-Time Applications with UML*. Addison-Wesley Professional, 2000.

[5] M. Jaragh and K. Saleh, "Modeling communications protocols using the unified modeling language," in *TENCON'2000, Intelligent Systems and Technologies for the New Millenium*, Kuala Lumpur, Malaysia, 2000.

[6] K. Kavi, D. Kung, H. Bhaambhani, G. Pancholi, M. Kanikarla, and R. Sah, "Extending UML to Modeling and Design of Multi-Agent Systems," in *International Conference on Software Engineering*, 2003.

[7] J. Wei, S. Cheung, and X. Wang, "Exploiting automatic analysis of e-commerce protocols," in *25th Annual Computer Software and Application Conference*, Chicago, Oct 2001.

[8] S. Lu, W. Halang, H. W. Schmidt, and R. Gumzej, "A component-based approach to specify hazards in the design of safety-critical systems," in *Industrial Informatics (INDIN'05)*, Aug 2005.

[9] "OMEGA project: Correct development of real-time embedded systems." [Online]. Available: http://www-omega.imag.fr

[10] SAE Standards, "Architecture analysis and design language (aadl), as5506," Society of Automotive Engineers, Tech. Rep., November 2004.

[11] S. Balsamo, M. Bernardo, and M. Simeoni, "Combining stochastic process algebras and queueing networks for software architecture anaysis," in *Proc. of the 14th Int. Conf. on Software Engineering and Knowledge Engineering (SEKE'02)*, A. Press, Ed., S. Angelo d'Ischia, Italy, 2002.

[12] L. Apvrille, J.-P. Courtiat, C. Lohr, and P. de Saqui-Sannes, "TURTLE: A Real-Time UML Profile Supported by a Formal Validation Toolkit," *IEEE transactions on software Engineering*, vol. 30, no. 7, jul 2004.

[13] D. Caromel, C. Delbé, A. di Costanzo, and M. Leyton, "ProActive: an integrated platform for programming and running applications on grids and P2P systems," *Computational Methods in Science and Technology*, vol. 12, no. 1, pp. 69–77, 2006.

[14] D. Caromel, L. Henrio, and B. Serpette, "Asynchronous and deterministic objects," in *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 2004, pp. 123–134.

[15] V. Mencl and M. Polak, "Uml 2.0 components and fractal: An analysis," in *5th Fractal Workshop - ECOOP'06*, Nantes, France, July 2006.

[16] "Common component modelling example (cocome)." [Online]. Available: http://agrausch.informatik.uni-kl.de/CoCoME

[17] H. Garavel, F. Lang, and R. Mateescu, "An overview of CADP 2001," *European Association for Software Science and Technology (EASST) Newsletter*, vol. 4, pp. 13–24, Aug. 2002.

[18] I. Černá, P. Vařeková, and B. Zimmerova, "Component substitutability via equivalencies of component-interaction automata," in *Proceedings of the Workshop on Formal Aspects of Component Software (FACS'06)*. Prague, Czech Republic: To appear in ENTCS, September 2006.

[19] A. Cansado and D. Caromel and L. Henrio and E. Madelaine and M. Rivera and E. Salageanu, *The Common Component Modeling Example: Comparing Software Component Models*, to appear 2007, http://agrausch.informatik.uni-kl.de/CoCoME.

[20] T. Barros, R. Boulifa, and E. Madelaine, "Parameterized models for distributed java objects," in *Forte'04 conference*, vol. LNCS 3235. Madrid: Spinger Verlag, Sept. 2004. [Online]. Available: http://hal.inria.fr/inria-00087222

[21] T. Barros, L. Henrio, and E. Madelaine, "Verification of distributed hierarchical components," in *International Workshop on Formal Aspects of Component Software (FACS'05)*. Macao: ENTCS, Oct. 2005.

[22] M. Morel, D. Caromel, and N. Parlavantzas, "Multicast and Gathercast interfaces for the GCM," Sophia Antipolis, France, October 2005, www.coregrid.net/mambo/content/view/188/30/.