16 A Specification Language for Distributed Components Implemented in GCM/ProActive

Antonio Cansado, Denis Caromel, Ludovic Henrio, Eric Madelaine, Marcela Rivera, and Emil Salageanu

INRIA Sophia-Antipolis, Université de Nice Sophia-Antipolis, CNRS, France
{first.last}@sophia.inria.fr

16.1 Introduction

16.1.1 Goals and Scope of the Component Model

This chapter is based on a component model for distributed components called *GCM* for *Grid Component Model*. We present here this component model, its reference implementation based on the Java middleware ProActive, our specification language, *JDC*, adapted to distributed software components, and the associated specification platform: *Vercors*. From the specification of components and their behaviour, our aim is to both verify properties of this behaviour and generate code for real GCM components.

The OASIS team works in the area of distributed systems, with a stress on programming methodology and tools for computational grids. We base our programming model on active objects: each active object has its own (and unique) thread, processing asynchronous method calls on the object.

Analysing the behaviour of distributed systems always has been more difficult than for traditional systems, because of the complexity of temporal interactions between remote parts of the application. However, the programming model proposed by our team (featured by the ProActive library [1] implementing the ASP calculus[2]) intends to ease the analysis of concurrency by clearly identifying entry points for communications and limiting the concurrency aspects to several remote method calls on the same object. Moreover, structuring distributed systems using components is somewhat better, because it requires interaction points (service points, but also required interfaces) to be well defined. Additionally, these well-defined boundaries reduce the amount of information that one component needs to have about its environment.

Unfortunately, developers still have to face non-trivial runtime incompatibilities when assembling off-the-shelf components. These arise due to the lack of a precise dynamic specification of the component behaviour (the protocols for inter-component communication). In fact, state-of-the-art implementations of component models such as Fractal [3] and the CORBA Component Model [4] only consider type-compatibility for binding interfaces.

Nonetheless, a much more precise, sound static compatibility check of bound interfaces can be achieved if behavioural information is added to the components. In such effort, we proposed in [5] to attach behaviour as part of the architecture

A. Rausch et al. (Eds.): Common Component Modeling Example, LNCS 5153, pp. 418–448, 2008.

specification, defined in terms of parameterized networks of transition systems. The compatibility between these behaviours can be checked using model-checkers as we did in [6]. In the same spirit, "behavior protocols" [7] and "component-interaction automata" [8] are ongoing research projects with similar goals. They both adopt a simpler model than ours, for example the "behavior protocols" framework uses a regular-language to describe traces of the component behaviour (without data), while we are able to take into account an abstraction of the data-flow which fulfils value-passing and routing.

One question that may rise is why to create a new specification language. Some of them, like the ISO language LOTOS, are very expressive both on the side of describing parallelism and synchronisation and on the side of data types. We argue that none of them fits well when it comes to distributed components (DCs), like the ones used in Grid computing to aid software design. In Grid computing, latency plays a major role, so asynchrony is a must for optimum resource usage. Additionally, designs usually exploit physical symmetries but are often obstructed by specification languages expressiveness. For example, in Fractal's standard Architecture Description Language (ADL), it is not possible to define *multiple components* (i.e., sets of identical components) and bindings between them in a natural way. Moreover, as DCs may be deployed over thousands of machines, collective interfaces have been defined as an extension of "classical" component models to address scalability of both designs and implementations. Last, most existing models focus only on the functional (business) behaviour, whereas we are interested in the non-functional behaviour as well. In Fractal and GCM vocabulary, non-functional behaviour includes mainly deployment, life-cycle and reconfiguration aspects such as replacing components and dynamically creating components. Within our models we are able to prove correctness of deployment and basic reconfiguration [9, 10].

In addition to the GCM component model and its implementation in ProActive, we propose here a new specification language called JDC (for Java Distributed Component) for answering requirements of DC applications, with stress on Grid programming. Part of this language may also be specified as UML 2 component diagrams and state-machines; we have developed a tool named CTTool for editing those diagrams and interfacing with verification tools. Being verifiable, the JDC specification can guarantee various safety properties in the derived implementation.

16.1.2 Contribution

In this chapter, we shall:

- introduce the Grid Component Model (GCM) and its reference implementation with the ProActive middleware (16.2.1 and 16.2.2), the JDC specification language (16.2.4), and the CTTool diagram editor (16.2.5);
- present each JDC feature with examples using small pieces of CoCoME;
- give an integrated example of one of the CoCoME components;
- present and comment the implementation of the CoCoME we realised with GCM/ProActive;

– explain how CoCoME requirements can be checked from the JDC specification, using our verification platform (16.5).

In addition, the case-study is available at our website[1], including the JDC specification, the CTTool diagrams, and a GCM/ProActive implementation.

16.1.3 Part of CoCoME Modelled

We did not model the complete CoCoME but focus on the CashDeskLine component, of which we gave a full and detailed specification. For the rest of the system (the inventory and the bank) we only provided simple components with a minimal behaviour, enough to act as an environment to the CashDeskLine. Throughout our approach we tried to stick with the original architecture as much as possible. This includes modelling the routing capabilities of the EventBus, and multiplicity of CashDesks.

The CoCoME as given does not represent a usual GCM design, and therefore does not profit from most of its features. For example, in Fractal and GCM there are no such thing as *Channel* components. They were modelled as a "normal" GCM component which is costly in performance and in the number of communications: the EventBus works as an intermediate component which replicates the calls as a router. More specifically for the ProActive/GCM implementation, in the CoCoME most of the calls are `void` which do not take advantage of the future mechanism, but only of the asynchronous calls featured by our programming model. We could have restructured the system with the GCM in mind, but it would have been a different design, unmatched with the rest of the book.

16.1.4 Benefits

Our methodology is based on a rich specification of hierarchical and distributed components. In fact, data is considered explicitly inside the behavioural model. This allows us to reason about value-passing properties as well as data used in routing within parameterized components.

These specifications can be verified against functional properties, as well as non-functional properties, expressed as branching-time temporal properties. We support the modelling with either the JDC language, or its graphical subset in UML, and the generation of behavioural models based on the specification.

The JDC developer starts by specifying the system behaviour and architecture, then checks properties and verifies coherence of his components, and finally generates code-skeletons with the control code of his application. Then, this generated code must be completed with the detailed business code, that should be written by the programmer, with the constraint that it does not affect the behaviour. We expect this technique to be applicable by non-experts on formal methods, specially when using our UML 2 graphical modelling tools.

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

16.1.5 Lessons Learned

Our objective when modelling the CoCoME was to show the effectiveness of our approach for modelling and verifying the behaviour of a medium-scale example, not particularly fitted to the GCM (as explained in Section 16.1.3 above).

The modelling allowed us to stress our tools with a medium-scale case-study, and we were able to generate the full state-space for a synchronous version of CoCoME. An asynchronous model would generate a much bigger state-space; however for a large class of temporal formulas (typically reachability, but not absence of deadlocks) we are able to use on-the-fly techniques which do not require brute-force generation of the state-space. The conclusion of this chapter details the current limitations of our tool platform.

16.2 Component Model

This section first presents the GCM and its ProActive implementation. Then we give a short description of our low-level semantic model, called pNets [11]. Then we describe the JDC specification language, giving its syntax and an informal semantics, and finally present the CTTool diagram editor.

16.2.1 The Grid Component Model (GCM)

The Grid Component Model (GCM) [12] is a novel component model being defined by the european Network of Excellence CoreGrid and implemented by the EU project GridCOMP. The GCM is based on the Fractal Component Model [3], and extends it to address Grid concerns.

Fractal is a hierarchical component model, meaning that every component is either a composite, i.e. it is composed of other components (and thus its content is known), or primitive if its content cannot be decomposed. Fractal components interact through interfaces which correspond to object interfaces, so interactions between components are method calls. *Server interfaces* receive invocations, whereas *client interfaces* send invocations. A general view of a composite component is seen in Figure 1. Interfaces on the left side are *server interfaces*, on the right side *client interfaces*, and on top *non-functional server interfaces*.

From Fractal, GCM inherits a hierarchical structure with strong separation of concerns between functional and non-functional behaviours. For example, it defines non-functional interfaces managing the application life-cycle and deployment. GCM also inherits from Fractal introspection of components and reconfiguration capabilities, mainly consisting in the possibility to change the content of a composite component and the bindings between components. Grids consider thousands of computers all over the world, for that, GCM extends Fractal using asynchronous method calls for dealing with latency. Grid applications usually have numerous similar components, so the GCM defines collective interfaces which ease design and implementation as they provide some synchronisation and distribution capacities. There are mainly two kinds of collective interfaces in the GCM: multicast interfaces that distribute one message with its parameters to a

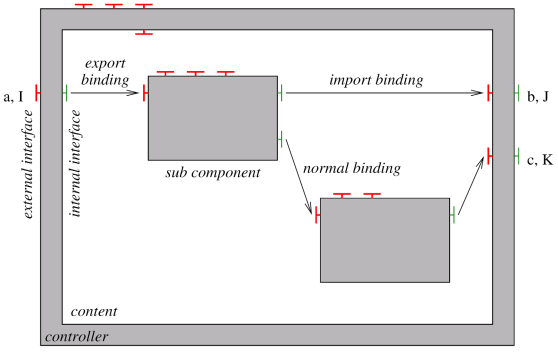


Fig. 1. A composite component as defined in Fractal Specification

set of destinations; and gathercast interfaces that synchronise and gather a set of messages with their parameters. A client interface may be a multicast interface, meaning that a call toward this interface can be distributed to many server interfaces depending on the distribution used. Similarly, a server interface may be a gathercast interface, meaning that multiple client calls will be synchronised and a single call will be performed towards the service component.

For adaptivity purposes, the GCM first extends the reconfiguration capabilities of Fractal to the non-functional aspects: the control of a component can be reconfigured dynamically, moreover, the GCM specifies interfaces for the autonomic management and adaptation of components.

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 also the definition of so-called *virtual nodes* (VN) 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.

16.2.2 A GCM Reference Implementation: GCM/ProActive

A GCM reference implementation is based on ProActive [1], an Open Source middleware. 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. 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.

To ensure causal ordering of requests, method calls use a rendez-vous protocol: the caller is momentarily blocked while requests are en-queued in the callee (server) side, and a future is created as a placeholder for the returned result. Then, the caller's 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 in a mechanism called *wait-by-necessity*. Therefore, there is an implicit data-flow synchronisation generated by the flow of futures. A precise operational semantics of ProActive is given by the ASP-calculus [2, 13]; it allows the proof of generic and important properties of ProActive's constructs on top of which we base our specification language.

Each primitive component is associated to an active object that should be written by the programmer (or generated by a JDC specification, as we will see in the following), whereas the active object managing a composite is generic and provided by the GCM/ProActive platform. Like in Fractal, GCM/ProActive component interfaces are realised by object interfaces. The particularity of GCM/ProActive is that those method calls are necessarily asynchronous, a method call on a server interface adds a request to its *request queue*. If this component is a composite, then it should simply forward the request to the contained component that is connected to this server interface. If the component is a primitive, then the required functionality should be addressed by the active object encapsulated in the primitive, so the request will be treated locally. Most of the time, requests are served in a FIFO order but any *service policy* can be specified when programming active objects (that is for primitive components), by writing a specific method called `runActivity()`. 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. One particularity of this approach is that it unifies the concept of component with the unit of distribution and parallelism.

Indeed, one essential property of ProActive, inherited from the ASP calculus, is that the global behaviour of a ProActive application is totally independent of the physical localisation of active objects and components on a distributed architecture. This allows us to totally separate the deployment file from the system specification: the deployment is one part of the implementation, and is isolated in a specific file so that the application need not be changed when run on different infrastructures (even going from a single JVM on one machine to a full multi-site heterogeneous grid of hundreds of machines).

16.2.3 pNets and Behavioural Models

In a previous work [11] we introduced "Parameterized Networks of Transition Systems" (pNets), as a powerful model and a generic low-level formalism, able to express communicating processes with value passing and parameterized topologies of processes. Parameterized synchronisation vectors allow to encode many different synchronisation and communication mechanisms amongst asynchronous processes.

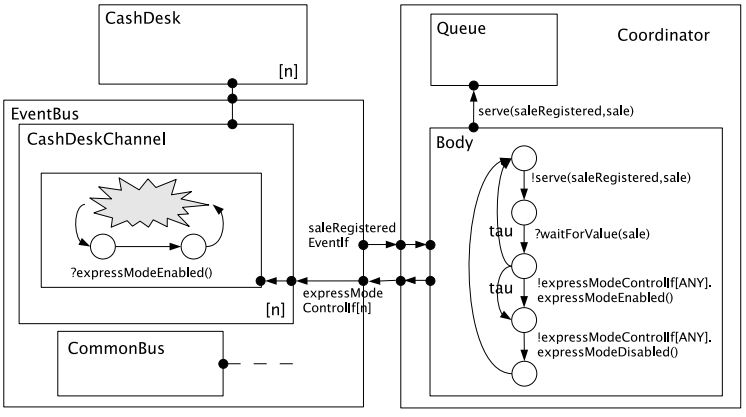


Fig. 2. A partial pNets model of CoCoME

We have shown how to describe the behaviour of ProActive's active objects with pNets. Then, in [9, 10] we synthesised a behavioural model for GCM/ProActive components, including their functional and non-functional behaviour. In Figure 2, we provide a partial model of CoCoME in pNets. Components are represented as parameterized processes, n being a parameterization over the number of CashDesks and of CashDeskChannels for representing multiple components. Note that this parameter is also used to provide multiple ports (collection interfaces in GCM language). Transitions encode communication with value passing, and internal actions (tau) provide both the hiding mechanisms necessary for component visibility rules, and the non-determinism required by specification-level abstractions. Other processes appear as additional transition systems, for example the *Queue* which encodes part of the ProActive's behaviour. Then, the automaton representing the Coordinator component is explicitly given: first it synchronises with the Queue for serving requests; then synchronises with the argument sent; finally it may non-deterministically restart its functional behaviour, enable a Cashdesk, or disable a CashDesk. For that, it synchronises with an arbitrary port within `expressModeControlIf` with the ANY keyword. Note that this port is parameterized and the index is used to address a specific CashDeskChannel. We do not go deeper into the pNets model in here, however the interested reader may refer to [11].

In each of these applications, the models express potentially infinite automata. In theory this allows us to apply many different proof methods, including inductive theorem proving techniques, state-of-the-art engines implementing decision procedures for decidable fragments of predicate logics or for regular structures. In practice, the current version of our verification platform only uses explicit-state model-checkers. For those, we first transform the pNet model into a network of finite Labelled Transition Systems (LTSs), using abstractions based on finite partitions of the parameter domains.

The pNet formalism is too low-level to be used directly as a specification language, and lacks the high-level concepts particular to the different context in which we want to use it. But in [10] we have shown that it plays an important role as an intermediate format between the platform tools; it can be viewed as the low-level semantic notation expressing both behaviour, structure and synchronisation of our components. This pNet model is the main intermediate format used by our tools to interface with the verification tools (see Section 16.4.1).

16.2.4 The Java Distributed Component Specification Language (JDC)

In order to bridge the gap between an architecture specification and a Java implementation, we have defined JDC for specifying distributed components. The JDC is meant to be used by developers at the early stages of conception of a component or of an application. A JDC specification will be used:

- on one hand to check the correctness of the component structure: model-checking against user requirements or scenarios, and against the global black-box specification of composite components, correct assembly of components,
- on the other hand to generate ADL definitions for the composite components and Java skeleton code for each of the primitive component, in which the developer will only need to fill-in the body of service methods.

In the remaining of this section we describe the syntax of JDC, introducing one by one its main concepts (but without going too deep in the details of the grammar). For each construct we refer to lines of code within the following sections in which an example can be found.

Components: JDC's main construct reflects the structure of a component: a component declaration comprises a part defining its external view, that is the definition of its interfaces and the specification of the black-box behaviour (interaction between services). For composite components, the declaration also comprises the definition of its architecture, that is the declaration of subcomponents and of their bindings. See an example in lines 60-66, page 433.

In addition, standard Java files are used to define the component interfaces signatures (as Java interfaces), and user-defined data-types (as Java classes).

In the following abstract syntax, the interface section is mandatory, the services section is mandatory for primitives and optional for composites, the architecture section is mandatory for composites. Remark that a component can be seen as a primitive in a JDC specification, and later be implemented by a composite. In that case the behavioural specification of this component will allow for a formal check of compatibility (or substitutability in the vocabulary of [8]).

```
1 component <NAME> (<Params>) {
    interfaces
3     <interface> *
    services
5     <service> *
    architecture
```

```
7       contents
          <component> *
9       bindings
          <binding> *
11  endComponent}
```

Interfaces: Interface definitions come in two parts: the first is a declaration within the JDC component structure, specifying each server (provided) and client (required) interface of the component, together with its type, its name, and optionally its parameter list. See lines 62-64.

```
12  <interface> := (server|client) interface <InterfaceType> <InterfaceName>[<Params>]
```

The second part is a standard Java Interface that defines the interface type, including the list of methods with their signature. This corresponds more generally to the Interface Definition Languages (IDLs) of some other component frameworks. See lines 67-78.

Services and policies: This section of the component specification defines the behaviour acceptable at the interfaces of the component. That is to say, how the component provides services when viewed as a black-box. This includes the required protocols, expressed as temporal ordering of the interactions (method calls) on the interfaces, but also the dataflow induced by the parameters of those methods, and that can be relevant to the behaviour logics. See lines 79-114.

```
    <service> =
14        service "{"
            <Local Variable Declaration> *
16          policy "{" <RegularExpression> "}" *
            <Service Method Declaration> *
18          <Local Method Declaration> *
            "}"
```

The protocols themselves are (non-deterministic) state-machines, expressed using regular expressions. Their events are either service instructions, or direct local method calls. The service instructions allow the user to specify, depending on the internal state of the protocol, which kind of methods to select, and in which order to pick them from the queue. See lines 81-85 and 201-209.

```
20  <regularExpression> =
       <serveMode> "(" <filter> * ")"
22    | <Method Call>
      | <regularExpression> ";" <regularExpression>
24    | <regularExpression> "|" <regularExpression>
      | <regularExpression> "*"
26
    <filter> = <InterfaceName>
28         | <InterfaceName> "." <MethodName>
```

Then for each method in each server interface, we need a method declaration, that will capture at least the relevant dataflow between input parameters and results of the method. The method body contains Java code, with eventual calls to client interface methods, and possibly a special "oracle" function called `__ANY(..)` that non-deterministically returns an arbitrary value of a given class. Service methods and local methods are not authorised to access the request queue. Their bodies consist of standard Java syntax (we do not provide a grammar for them). See e.g. lines 209-223.

Subcomponents and bindings: In this part of the specification, we specify the subcomponents of a composite component, as instances of components (types) that may be defined separately (see lines 120-126), and bindings (lines 127-158) connecting subcomponent's interfaces with internal interfaces of the composite (import and export bindings in Figure 1), and connecting subcomponents together (normal bindings in Figure 1).

For grid structures, it is important to be able to have indexed sets of subcomponents within a composite, and to have syntax to specify their bindings in terms depending on their indexes. For that, parameters may be defined within the `for` statement (as `Params`), and used within the `bind` statement as a subset of these parameters in the `ActualParams`. See lines 134-136.

```
   <component> =
30    <ComponentType> "(" <Params> ")" <ComponentName>

32 <binding> =
      bind "(" <ComponentName>"."<InterfaceName>, <ComponentName>"."<InterfaceName> ")"
34   | for "(" <Params> ")" "{"
         bind "(" (<ComponentName> | <ComponentName> "[" <ActualParam> "]" ) "."
36                (<InterfaceName> | <InterfaceName> "[" <ActualParam> "]" ) ","
                  (<ComponentName> | <ComponentName> "[" <ActualParam> "]" ) "."
38                (<InterfaceName> | <InterfaceName> "[" <ActualParam> "]" ) ")"
      "}"
```

Exceptions: For dealing with latency, GCM provides asynchronous communications and exception handling. Exceptions typically influence the control flow of the application, and therefore are important to consider within a JDC specification. JDC allows for asynchronous method calls with exceptions, but with some restrictions: the control flow may not leave the `try` block where the exception should be thrown. For that, before leaving a `try`, all pending exceptions act as a barrier until its safe execution path is known. Additionally, a future which result comes from a method with exceptions cannot be sent as argument to another component, i.e., it is implicitly blocked before calling the method until the return value is known. Below, we show the abstract syntax of both method with exception definition and a control block. The reader should find it familiar to Java. See lines 102-112.

```
40    <ReturnType> <MethodName>(<Params>) throws "(" <ExceptionType> ")" *

42    try "{"
          <UserCode> *
44    "}"
      catch "(" <ExceptionType> <ExceptionName> ")" "{"
46        <UserCode> *
      "}"
```

Parameters and collective interfaces: Another feature dealing with GCM is collective interfaces. There are multicast and gathercast interfaces for providing distribution and synchronisation of data. In JDC, these can be defined both at a black-box level or at an architecture level. Moreover, the designer also defines the parameter distribution that applies. For example, a multicast interface may

broadcast a same method call to multiple components. The abstract syntax is as follows:

```
48    multicast "(" <ComponentName>"."<InterfaceName>"," <DistributionType> ")" "{"
          bind "(" <ComponentName>"."<InterfaceName>"," <ComponentName>"."<InterfaceName> "
              )" *
50    "}"
```

Data abstraction: Apart from the specific component structure constructs, the JDC syntax is intentionally very close to the Java language. This is mainly because all code within the JDC specification uses real Java datatypes to keep compatibility with Java. However, if one wants to verify the specification (using model-checking or equivalence-checking), it is mandatory to provide a data abstraction of every user-class used in the specification into (first order) Simple Types. This is particularly important to avoid usual state-explosion during the model-checking of the system, and to allow exhaustive search of execution paths. Moreover, Simple Types are the only datatypes accepted by pNets, for more fundamental reasons: this is the basis for the preservation of safety properties by finite abstraction in our verification method. So every class in the JDC specification must be mapped towards Simple Types, and this mapping must be an abstract interpretation. See an example in lines 224-240.

These are given in Java classes and may be used within a JDC specification. Concretely, they are:

- integers, enumerated types, strings, booleans;
- intervals of integers;
- records of simple types;
- arrays of simple types.

```
      class <ClassName> "{"
52        (public | private | protected)":" *
          <ClassType> <FieldName>";" *
54        <ClassType> <FieldName> abstracted as <SimpleType>";" *
          <ReturnType> <MethodName> "(" <Params> ")" <Exception>";" *
56        <ReturnType> <MethodName> "(" <Params> ")" <Exception> abstracted as "{"
              <UserCode> *
58        "}" *
      "}"
```

In the abstract syntax above, data abstraction is done over all methods and attributes used within the JDC specification. For these, the keyword `abstracted as` is used to map whatever method or attribute into Simple Types. In the method body, usual JDC code can be used as far as the final result is a Simple Type, and all data used is mapped to Simple Types as well. Other methods and attributes not used within the specification can be left unspecified as they are not considered by neither code generation nor model generation.

16.2.5 UML Component Diagrams, and the CTTool Editor

The JDC is a textual language, certainly adequate for trained developers, and for specifying big and/or complex systems. However, for casual users, or for fast

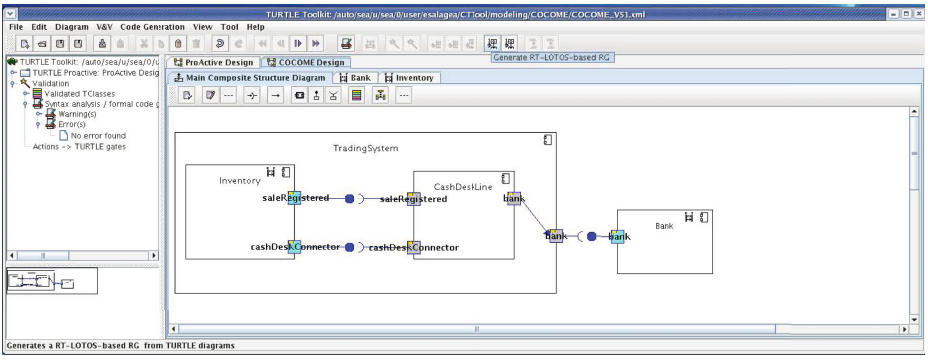


Fig. 3. The *CTTool* UML2.0 modelling software

design in the first steps of the development cycle, one may prefer a graphical language. With this goal, we have built a UML editor, called CTTool (Figure 3), dedicated to component specification, and providing an integrated interface with a model generator and a model checker.

The main constructs can be found in Figure 4.

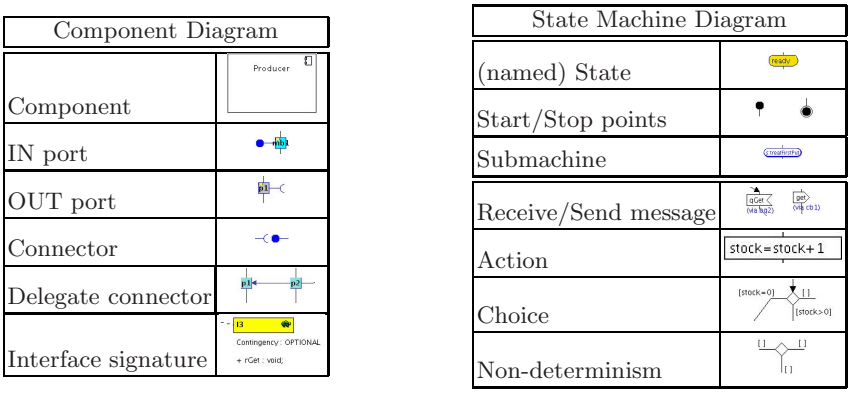


Fig. 4. Component and State Machine Diagram Constructs

CTTool was developed starting from a similar software named TTool, developed by a research team at ENST-Paris [14]. TTool is more specifically dedicated to the design and verification of embedded systems, or systems on chips. Compared with TTool, CTTool features UML2.0 diagrams (component and state-machines) with hierarchical designs (Fig. 4); it inherits from the connection of TTool with the CADP verification tools [15], through generation of LOTOS code.

The current version of CTTool is an intermediate step, in the sense that it has no primitive for specifying distributed features of our components, and consequently no Grid specific primitive. Moreover the generated code has the

same synchronous semantics than the original TTool, so the proofs that one can conduct in CTTool are only for synchronous components systems, rather than for the asynchronous semantics of GCM/ProActive including asynchronous requests and future values. We shall explain how this has influenced the verification of the CoCoME scenarios in section 16.5.1

Still we found usefull to provide CTTool drawings, together with the corresponding JDC code in the following pages. In future versions, CTTool will have specific high-level constructs for distributed Grid applications (service policies, multicast and gathercast communications, etc). It will certainly be re-engineered to produce JDC code, or at least generate the proper asynchronous semantics in the constructed models.

16.3 Modelling the CoCoME

Including in this chapter the specification of the full CoCoME case-study would not be possible. We have chosen to include only the specification of two components, with the required views of each, with the JDC text, and some of the CTTool diagrams as illustrations. The first one is the (composite) CashDesk component (Chapter 3, Figure 13), for which we include and explain:

- its black-box view with the definition of external interfaces, and the specification of its visible behaviour;
- its architectural view with its subcomponents and bindings;
- its GCM view, with excerpts of the generated ADL code.

The second one is the (primitive) CashBoxController component, with:

- its black-box view with its interfaces and behaviour, with much more details on the definition of its service methods;
- no architecture as it is primitive;
- an abstraction specification of a user-defined datatype;
- pieces of generated Java/ProActive code;
- a fragment of its deployment specification.

Before starting with those examples of component specifications, let us define what are the different views that we use.

16.3.1 Black-Box View

We call black-box view of a component its externally visible architecture and behaviour. Therefore it includes the common part of both primitive and composite component: list of its interfaces (defining which are client and server interfaces), and definition of these interfaces (the Java methods and their signatures). The black-box view allows to use the component without knowing anything of its internals. This is usual in many programming languages when thinking simply of (static) typing of the interfaces; but here we aim at dynamic compatibility of components, so we need a precise enough specification of the protocol of its

interactions with other components, namely the temporal ordering of activation of service methods, and of calls to external (client) methods. In such a protocol we include the synchronisation, control flow and data flow of the system, as observed during communications between components.

In [10] we have shown how to specify both the functional and the non-functional behaviour of GCM/ProActive components; however for the time being we focus only on the functional (also known as business) behaviour, leaving the non-functional parts (life-cycle, bindings, reconfiguration management) to further developments of the JDC.

16.3.2 Architectural View

The architectural view gives a one-level refinement of a component as a composition of subcomponents. For each one of these subcomponents, the designer must provide its black-box view. Note that, when an architecture is provided for a component, the `policy` section in the black-box definition is optional as it is implicitly defined by the architecture.

From the user point of view, the architecture specification is a functional delegation to subcomponents, similar to what the GCM ADL stands for. In it, subcomponents and internally visible bindings of a component are defined (see Figure 1); bindings can be either between the parent component and one of its subcomponent (export binding), between a subcomponent and the parent (import binding), or bindings between subcomponents (normal binding).

We did not use the GCM ADL for defining the architecture because we want to provide more expressive power than usually expressed within an ADL. Typically, one may want to define dynamic architectures, i.e., systems with dynamic instantiation of components. These kind of architectural specifications are not meant to be captured by the GCM ADL, though they might be possible by extending the language.

For the CoCoME model, we have specified the architecture, whenever given in the CoCoME reference, of every component except for the Inventory. The latter was given only a black-box specification to simplify the model. Within this section we show the architecture of the CashDesk component.

16.3.3 GCM View

Our objective is to generate GCM components from the JDC specification. JDC is rich enough to be able to generate automatically both the ADL describing the structure of the application, and the skeleton of the primitive components in ProActive.

For the composite components, the ADL can be automatically inferred from the architectural view presented above. For the primitive components, the ADL mainly consists in the definition of interfaces and can be generated automatically. The skeleton of the Java class implementing the primitive can be generated from the JDC black-box specifying the behaviour of the component. The user then only has to write the business code, resolving all the abstractions and non-determinism present in the black-box definition.

Concerning non-functional aspects, they are not specified for now in the JDC. Thus it is also the role of the programmer to provide and compose these aspects. In general, most of those aspects are provided by the component middleware, e.g. Fractal requires basic management controllers to be implemented by each component. In most cases, dealing with non-functional aspects consists in invoking operations on these controllers. For the moment, those invocations have to be performed manually by the programmer; but on the long term basis we would like to include them in the JDC so that it is possible to study and verify the interaction between functional and non-functional concerns.

The generation of code from the JDC specification is not working yet; thus based on the JDC specification we wrote the GCM/ProActive code for the CoCoME example. This code consists of a set of composite components defined in the ADL, and a set of primitive components written in Java and using ProActive. Recall that, at deployment, a thread is created for each primitive and each composite component, and those components communicate by asynchronous method calls with transparent futures, leading to a parallel and distributed implementation of the application.

16.3.4 Deployment View

The deployment view of an application is out-of-scope for the JDC specification as we leave it as a middleware concern. However, since the modelling of distributed aspects is an important part of a specification, this section outlines the ProActive deployment scheme, and shows how the distribution nature of GCM components can be captured. Further, using the CoCoME architecture as an example, it is shown how to map the components to the physical infrastructure.

ProActive and GCM comprehensive deployment framework is based on the concept of *Virtual Node* (VN). A VN is above all an *application abstraction* that, at modelling or programming time, captures the distributed nature of a system. Typically, a given application is specified to deploy on several VNs (e.g. each component on a separate VN), each capturing a specific entity or related set of entities of the application. At deployment, each VN is mapped to one or several machines on the network, using appropriate protocols.

The number and characteristics of VNs are chosen by the application designer, providing both guidance and constraints to be used and enforced at deployment time. Both parallelism and de facto distribution can be captured by VNs. Moreover, the designer can also specify multi-threaded constraints by using a single VN for several components, capturing a forced co-allocation. When building composite components, one has the possibility to merge some inner VNs into a single one, specifying co-allocation of the corresponding inner components. One also has the possibility to maintain at the level of the composite some of the inner VNs, specifying an independent mapping of the corresponding inner components to the physical infrastructure.

A VN has several characteristics. The most important is its *cardinality*, which can be *single* or *multiple*. The former captures the fact that a single node of the infrastructure has to be used to execute the corresponding component, the later

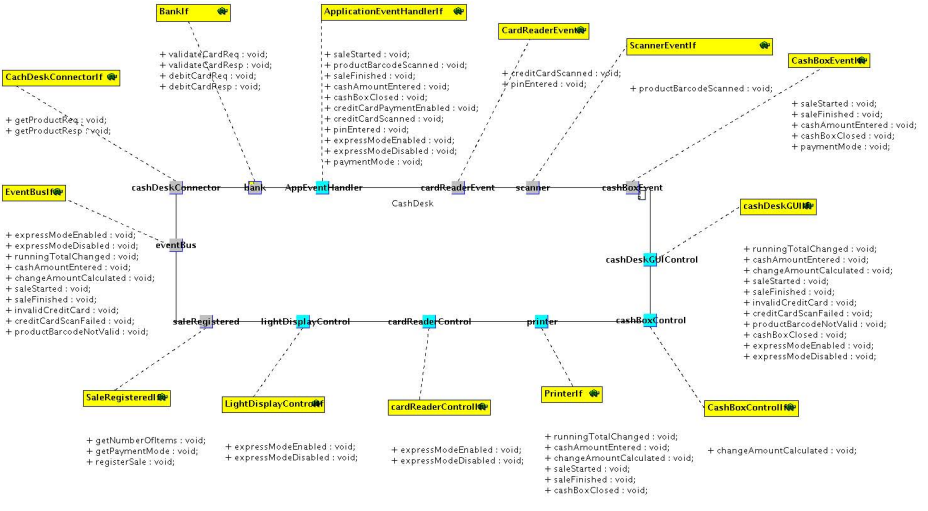
gives the possibility at deployment to map the component on several machines. This powerful possibility is to be related to multicast and gathercast interfaces: a collective interface often corresponds to a Multiple VN with the same cardinality.

At deployment time, all the VNs of the CoCoME specification will be mapped to one or several machines of the physical infrastructure using an XML file. The ProActive implementation makes it possible to choose from many protocols to select and access the actual nodes (rsh, ssh, LSF, PBS, Globus, etc.), and to control the number of components per machine, and per process (JVM).

16.3.5 Specification of the CashDesk Component

In this subsection, we specify the CashDesk component. We give its black-box specification together with a matching architecture specification. Finally, we outline its implementation within the GCM/ProActive.

Black-Box View of the CashDesk. The black-box of the CashDesk starts by defining its server and client interfaces. In there, we see that the `bankIf` is a collection interface as it addresses multiple banks, but method calls are routed to one bank at a time. Note that CTTool's diagrams do not have the definitive method signatures as type-checking is still limited within the tool.

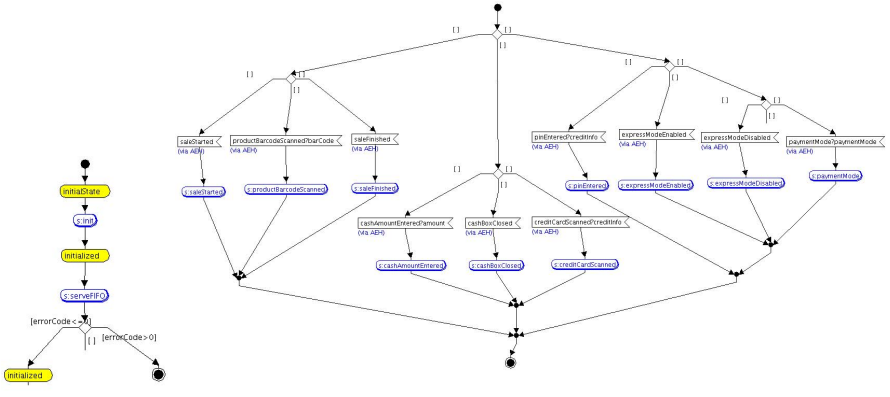



```
60 component CashDesk(int numOfBanks) {
     interfaces
62     server interface CardReaderControlIf cardReaderControlIf;
       server interface ApplicationEventHandlerIf applicationEventHandlerIf;
64     client interface BankIf banksIf[numOfBanks];
       // ... all 10 other interfaces
66 }
```

These interfaces must be properly defined. For example, in the code below we see the definition of the `ApplicationEventHandlerIf` interface, which exposes the full method signature using user-classes.

```
public interface ApplicationEventHandlerIf {
    void saleStarted() throws NotIdleException;
    void saleFinished() throws SaleNotFinishedException;
    void cashAmountEntered(CashAmount moneyAmountEntered) throws NotPayingException,
         WrongPaymentModeException;
    void cashBoxClosed() throws NotPayingException, WrongPaymentModeException;
    void creditCardScanned(CreditCardScanned creditCardScanned) throws
         NotAcceptingCreditCardException, WrongPaymentModeException;
    void pinEntered(PIN pin);
    void paymentMode(PaymentMode paymentMode) throws WrongPaymentModeException;
    void expressModeDisabled();
    void expressModeEnabled();
    void productBarcodeScanned(ProductBarcode barcode) throws ExceededNumberOfProducts;
}
```

No matter how the component is to be implemented (either by a primitive or a composite component), in JDC it is mandatory to provide a behavioural specification, either in the form of a black-box definition, or in the form of its architectural implementation, or both. For this component, we provide a black-box specification; this starts with the service policy defined with a regular expression. In the case of the `CashDesk`, there are multiple services denoting that there are multiple processes visible from the outside. This stands for a compact representation of the interleavings admitted by the component.



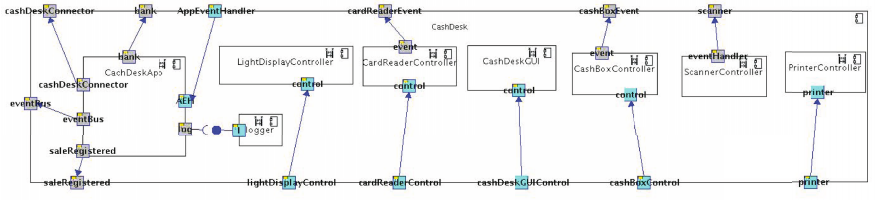

```
services
  service { // CardReaderController
      policy {
          ((emit() | serveOldest(cardReaderControlIf.expressModeDisabled))*;
          serveOldest(cardReaderControlIf.expressModeEnabled);
          serveOldest(cardReaderControlIf.expressModeDisabled))*
      }
      void emit() {
          if (__ANY(bool)) // non-deterministic choice
              cardReaderEventIf.creditCardScanned(__ANY(CreditCard));
      }
      // ... cardReaderControlIf.expressModeEnabled
      // ... cardReaderControlIf.expressModeDisabled
  }
  service { // CashDeskApplication
      locals {
          CashState cashState;
          // ... other local variables
      }
```

```
98          policy {
                init(); expressModeDisabled();
100             serveOldest(applicationEventHandlerIf) *
            }
102         void applicationEventHandlerIf.saleStarted() throws NotIdleException {
                switch (cashState.getState()) {
104                 case cashState.IDLE:
                        cashState = cashState.STARTED;
106                     break;
                    case cashState.STARTED:
108                 case cashState.PAYING:
                        throw new NotIdleException();
110                     break;
                }
112         }
            // ... other methods
114     }
    }
```

Note that each service defines its own service methods and possibly its own local variables.

In Section 16.3.6 we illustrate different aspects of the behavioural specification. Then in Section 16.4.2 we give example of code generated from the `CashDesk` service definition.

Architectural View of the CashDesk. As the black-box specification only defines the externally visible behaviour and not how it is implemented, there are, of course, several architecture definitions that match the same component's black-box specification. A possible architecture implementing the `CashDesk` is shown in the following. Note that we added a logger just to trap and print exceptions but it does not influence the component behaviour.



The first part of the architecture defines the subcomponents which compose the component. When applies, they are provided with parameters for their correct deployment as with the `CashDeskApplication`.

```
118 architecture CashDesk(int numOfBanks) {
        contents
120         component CashDeskApplication(numOfBanks) application;
            component CashReaderController cashReader;
122         component CashDeskGUI cashDeskGUI;
            component cashBoxController cashBoxController;
124         component LightDisplayController lightDisplayController;
            component PrinterController printerController;
126         component ScannerController scannerController;
```

Then, we define the bindings section exposing how the functional delegation takes place, and synchronisations between components. Note that it is possible to address either a specific component or a specific collection interface.

```
    bindings
        // application
        bind(this.applicationEventHandlerIf, application.applicationEventHandlerIf);
        bind(application.cashDeskConnectorIf, this.cashDeskConnectorIf);
        bind(application.eventBusIf, this.eventBusIf);
        bind(application.saleRegisteredIf, this.saleRegisteredIf);

        // bind all bank interfaces
        for (int i: numOfBanks) {
            bind(application.banksIf[i], this.banksIf[i]);
        }

        // cashReader
        bind(this.cashReaderControlIf, cashReader.controlIf);
        bind(cashReader.eventIf, this.eventIf);

        // cashDeskGUI
        bind(this.cashDeskGUIControlIf, cashDeskGUI.controlIf);

        // cashBoxController
        bind(this.cashBoxControllerControlIf, cashBoxController.controlIf);
        bind(cashBoxController.eventIf, this.cashBoxControllerEventIf);

        // lightDisplayController
        bind(this.lightDisplayControllerControlIf, lightDisplayController.controlIf);

        // printerController
        bind(this.printerIf, printerController.printerIf);

        // scannerController
        bind(scannerController.scannerIf, this.scannerIf);
}
```

The corresponding ADL file, in XML format, will be generated from this part of the specification. It will then be used both to generate the synchronisation structures for the model-checker, and as an input to the component factory of ProActive at deployment time, but all this is left as future work for the moment.

GCM View of the CashDesk. Now, we present a simplified ADL description of the CashDesk component. It focuses on the CashDeskApplication subcomponent, the other subcomponents being similar. The ADL description starts with the definition of the external interfaces of the CashDesk component, together with their roles (client or server).

```
<component name="CashDesk">
  <interface signature="CashDeskLine.if.LightDisplayControlIf" role="server" name="
       lightDisplayControlIf"/>
  <interface signature="CashDeskLine.if.CardReaderControlIf" role="server" name="
       cardReaderControlIf"/>
  <interface signature="CashDeskLine.if.CashDeskGUIIf" role="server" name="
       cashDeskGUIIf"/>
  <interface signature="CashDeskLine.if.CashBoxControlIf" role="server" name="
       cashBoxControlIf"/>
  <interface signature="CashDeskLine.if.PrinterIf" role="server" name="printerIf"/>
  <interface signature="CashDeskLine.if.ApplicationEventHandlerIf" role="server" name="
       applicationEventHandlerIf"/>
  <interface signature="if.CashDeskConnectorIf" role="client" name="cashDeskConnectorIf
       "/>
  <interface signature="if.SaleRegisteredIf" role="client" name="saleRegisteredIf"/>
  <interface signature="CashDeskLine.if.CardReaderEventIf" role="client" name="
       cardReaderEventIf"/>
  <interface signature="CashDeskLine.if.CashBoxEventIf" role="client" name="
       cashBoxEventIf"/>
```

```
170   <interface signature="CashDeskLine.if.ScannerEventIf" role="client" name="
          scannerEventIf"/>
      <interface signature="if.BankIf" role="client" name="bankIf"/>
172   <interface signature="CashDeskLine.if.EventBusIf" role="client" name="eventBusIf"/>
```

Then the subcomponent CashDeskApplication is described by its external interfaces, this component is a primitive one (line 180), so the path of its implementation is given (line 179).

```
      <component name="CashDeskApplication">
174     <interface signature="CashDeskLine.if.ApplicationEventHandlerIf" role="server" name
            ="applicationEventHandlerIf"/>
        <interface signature="if.CashDeskConnectorIf" role="client" name="
            cashDeskConnectorIf"/>
176     <interface signature="if.SaleRegisteredIf" role="client" name="saleRegisteredIf"/>
        <interface signature="CashDeskLine.if.EventBusIf" role="client" name="eventBusIf"/>
178     <interface signature="if.BankIf" role="client" name="bankIf"/>
        <content class="CashDeskLine.CashDesk.CashDeskApplication"/>
180     <controller desc="primitive"/>
      </component>
182   <component name="CardReaderController"> ...       </component>
      <component name="LightDisplayController"> ...     </component>
184   <component name="ScannerController">     ...      </component>
      <component name="PrinterController">     ...      </component>
186   <component name="CashBoxController">     ...      </component>
      <component name="CaskDeskGUI">           ...      </component>
```

Finally, the bindings of the CashDeskApplication are described, in this example only two kinds of bindings are shown: import bindings like the first one, and export bindings like the others.

```
188   <binding client="this.applicationEventHandlerIf" server="CashDeskApplication.
          applicationEventHandlerIf"/>
      <binding client="CashDeskApplication.cashDeskConnectorIf" server="this.
          cashDeskConnectorIf"/>
190   <binding client="CashDeskApplication.saleRegisteredIf" server="this.saleRegisteredIf"
          />
      <binding client="CashDeskApplication.eventBusIf" server="this.eventBusIf"/>
192   <binding client="CashDeskApplication.bankIf" server="this.bankIf"/>
      ...
194   <controller desc="composite"/>
    </component>
```

Deployment View of the CashDesk. When defining the CashDeskLine composite, a VN `CashDeskLineVN`, cardinality Multiple is specified as the composition of all the `CashDeskVN`. The effective cardinality of this VN is attached to the number of cashDesks (`numOfCashDesks` in the specification).

16.3.6 Specification of the CashBoxController Component

In this section, we specify the CashBoxController component. We give its black-box view, but the architectural view does not apply because we do not decompose the behaviour of the CashBoxController into subcomponents.

Black-Box View of the CashBoxController. The component has a client and a server interface, and defines a non-trivial service policy. The controller can be seen as an active component in the sense that it triggers events regarding the cashbox, so it is not awaiting for any signal to be received.

```
component CashBoxController {
  interfaces
      server interface CashBoxControlIf controlIf;
      client interface CashBoxEventIf eventIf;

  services
      service {
          policy {
              ( eventIf.saleStarted(); eventIf.saleFinished();
                ( cashMode(); cashAmount(); serveOldest(controlIf.changeAmountCalculated
                     ); eventIf.cashBoxClosed() )
                |
                ( creditCardMode() )
              )*
          }
```

There are no state variables (variables defined within the "locals" block), nevertheless the component is not stateless; the service policy implicitly defines that the component cycles through some states, each one defining which are the actions that the CashBoxController may do. For example, the component only serves requests from the queue when a client is paying with cash; otherwise, the component is seen as a machine sending events regardless of the environment (as the environment does not take the hardware interaction into account).

The component defines local methods and service methods; the latter have their method names prefixed by the interface they belong. For the method `changeAmountCalculated(..)`, and from the behavioural point of view, we are only interested in the access to the variable sent as argument, but not what we actually do with it; so the behavioural model can block the execution until the concrete value of the variable is known.

```
      // local methods
          void cashMode() {
              eventIf.paymentMode(new PaymentMode(CASH));
          }
          void creditCardMode() {
              eventIf.paymentMode(new PaymentMode(CREDIT));
          }
          void cashAmount() {
              eventIf.cashAmount(__ANY(CashAmount));
          }
      // service methods
          void controlIf.changeAmountCalculated(CashAmount changeAmount) {
              changeAmount.waitForValue();
      }}
  } // end of CashBoxController black-box definition
```

There are some non-deterministic choices, both in the events that may be sent, and in the data sent. For example, we do not know exactly which amount the user paid, so we use the "oracle" function `__ANY(CashAmount)` to choose any value within the variable domain. It will be up to the data abstraction to define this domain, or in the case of the implementation, to the programmer to define its real value. A mapping of this class into Simple Types is given as:

```
public class CashAmount {
    private double amount abstracted as enum {"ZERO", "NOT_ZERO" };

    public __ANY() {
        return new enum {"ZERO", "NOT_ZERO" };
    }
    public double getAmount() abstracted as {
```

```
        return amount;
    }
    public void add(CashAmount purchasePrice) abstracted as {
        if (this.amount == "ZERO" && purchasePrice.getAmount() == "ZERO")
            amount = "ZERO";
        else
            // non-determinism within the data abstraction as a negative
            // value could leave the amount in zero
            amount = __ANY(CashAmount);
}}
```

GCM View of the CashBoxController. Next, we give the implementation of the CashBoxController primitive component. This Java code is to be instantiated as an active object (in Java implementing the `RunActive` interface). The active object implements the `CashBoxControlIf` server interface and contains a field named `cashBoxEventIf` implementing the client interface of the component. Finally, it also implements Fractal's `BindingController` interface to allow dynamic binding of its interfaces.

```
public class CashBoxController implements CashBoxControlIf, BindingController,
     RunActive {
  public final static String CASHBOXEVENTIF_BINDING = "cashBoxEventIf";
  private CashBoxEventIf cashBoxEventIf;

  public CashBoxController () { } // empty constructor required by ProActive
```

Note that necessary hooks for Fractal controllers are implemented on the form of the four first methods of the object. These are generated automatically.

```
  // Implementation of the Controller interfaces
  public String[] listFc () {return new String[] { CASHBOXEVENTIF_BINDING };}

  public Object lookupFc (final String clientItfName) {
      if (CASHBOXEVENTIF_BINDING.equals(clientItfName))
         return cashBoxEventIf;
      return null;
  }
  public void bindFc (final String clientItfName,final Object serverItf){
      if (CASHBOXEVENTIF_BINDING.equals(clientItfName))
          cashBoxEventIf = (CashBoxEventIf)serverItf;
  }
  public void unbindFc (final String clientItfName){
      if (CASHBOXEVENTIF_BINDING.equals(clientItfName))
          cashBoxEventIf = null;
  }
```

Recall that `changeAmountCalculated` is the only method of the server interface, requests addressed to this component will be asynchronous calls to this method. It corresponds to the same method as in the black-box view above.

```
  // Implementation of the functional interfaces
  public void changeAmountCalculated(CashAmount changeAmount) {
      System.out.println(changeAmount.getAmount()); // the amount to be returned as
           change
  }
```

Then, the service method in ProActive (`runActivity`) implements the service policy. It is the kernel of a ProActive component as it exposes the component's behaviour, and is called by the middleware when the component is started. It consists of a set of invocations on the client interface (`cashBoxEventIf`), together with a blocking service on the `changeAmountCalculated` method. This method

is a direct translation of the policy section of the black-box presented above and should not be modified by the ProActive programmer.

```
  public void runActivity(Body body) {
      Service service = new Service(body);
      while (body.isActive()) {
          cashBoxEventIf.saleStarted();
          cashBoxEventIf.saleFinished();
          if ((new AnyBool()).prob(50)) {
              cashMode();
              cashAmount();
              service.blockingServeOldest("changeAmountCalculated");
              cashBoxEventIf.cashBoxClosed();
          } else
              creditCardMode();
  }}
```

By contrast the service method and the local (private) methods that are declared in the JDC (`CashBoxController` service declaration will contain the true functional code of the component, and are directly modifiable.

```
  private void cashMode() {
      cashBoxEventIf.paymentMode(new PaymentModeImpl(PaymentModeImpl.CASH));
  }
  private void creditCardMode() {
      cashBoxEventIf.paymentMode(new PaymentModeImpl(PaymentModeImpl.CREDIT));
  }
  private void cashAmount() {
      cashBoxEventIf.cashAmountEntered(new CashAmountImpl(1000)); // the client paid
           1000
  }
} // end of CashBoxController implementation
```

Deployment View of the CashBoxController. If we want to model a system where a `cashBoxController` can be instantiated on its own processing element, then a VN, for instance named `cashBoxControllerVN`, cardinality `single` has to be specified and exported at the level of the CashDesk. At the next level, when building the CashDeskLine component, the middleware will be able to take care of transforming this VN into an appropriate VN at the composite level.

```
<exportedVirtualNodes>
    <exportedVirtualNode name="cashBoxControllerVN">
      <composedFrom>
          <composingVirtualNode component="this" name="cashBoxControllerVN"/>
      </composedFrom>
    </exportedVirtualNode>
  </exportedVirtualNodes>
  <content class="CashDeskLine.CashDesk.CashBoxController"/>
  <virtual-node name="cashBoxControllerVN" cardinality="single"/>
```

16.4 Transformations

In this section we first describe the software tools that we are building for editing and analysing JDC specifications and CTTool diagrams, and generating ProActive code skeletons. Then we explain on an example how we intend to generate "safe by construction" code.

16.4.1 Tools Overview

In Figure 5 we sketch the general architecture overview of the specification and analysis platform that we are building, so-called Vercors. This figure does not show the architecture of the current version of CTTool, that contains its own model generation engine, and its own bridges [14] with the CADP provers through intermediate LOTOS code.

A JDC specification can be edited directly in a text editor, or generated from the diagrams of CTTool; but we also plan to develop an Eclipse plug-in that will ease the JDC development, and give an integrated interface to the various analysis and generation functions of the platform. For the moment, JDC has been conceptually defined, but does not have any tool support yet, so the description in this chapter is mostly left as mid-term future work. However, the synchronous version of CTTool is operational and available at our website.

The first part of the platform deals with data abstraction: data types in a JDC specification are standard, user-defined Java classes, but those must be mapped to Simple Types before generating the behavioural models and running the verification tools. The abstraction is part of the JDC syntax, but the tool will offer guidance to help define correct mappings that are correct abstract interpretations. This phase ends-up with a "JDC Abstracted Specification" in which all data are Simple Types, that are provided as a predefined library.

Such an Abstract Specification is then given as input to our model generator ADL2N [6], that implements the behavioural semantics of the language, and builds a model in terms of pNets, including all necessary controllers for non-functional and asynchronous capabilities of the components. Note that the current version of ADL2N works on ADL and LOTOS files rather than JDC. Additionally ADL2N implements a second step of abstraction, that uses finite partitions of the Simple Types to build finite models for classical (explicit-state) model-checkers. Potentially this step could be automatically derived from the syntax of a formula. In any case the pNets objects are hierarchical and very

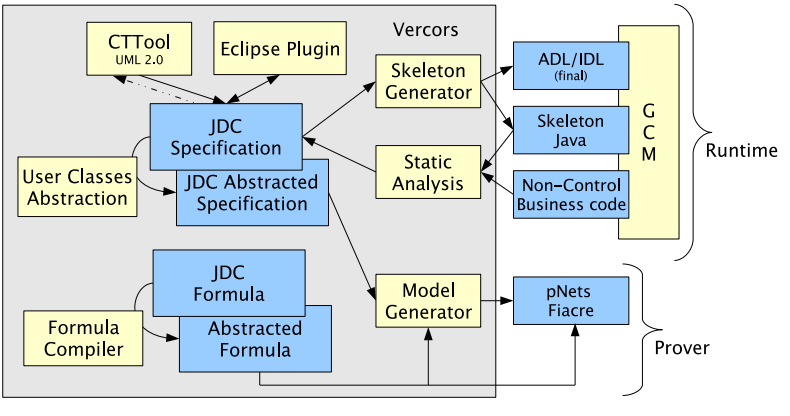


Fig. 5. The Vercors platform

compact as in pNets datatypes are kept parameterized – not instantiated – as well as families of processes (for multiple components).

Just like CTTool, our Vercors platform is using the CADP toolset [15] for state-space generation, hierarchical minimisation, (on-the-fly) model-checking, and equivalence checking (strong/weak bisimulation). The JDC property specifications will also be subject to the same abstractions, and will be translated into regular μ-calculus formula to be fed into the model-checker. In the future, we 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.

The upper right part of the figure is dedicated to code generation, that will be detailed in Section 16.4.2. The ADL files in XML syntax, the Java interface declarations, and all the Java code that constitute the control part of a component (the implementation of the service policies) will be generated. The method declarations for the service methods are created too, they can be then filled-in by the developper. This constitutes a runnable ProActive component or full application.

16.4.2 Generation of Safe GCM/ProActive Code

Although the code generation is not implemented yet, we give in here its main constructs. As an alternative to static analysis methods for proving that a user-implemented component conforms to its JDC specification, we rely on the generation of an ADL model plus GCM/ProActive code-skeletons from the specification. Then, we set rules stating what the user can do in the remaining code in such a way that the generated code will be guaranteed to be correct. We base our method on the following steps:

- the GCM ADL definition is automatically generated from JDC as it includes the architecture, interface signatures and component cardinality;
- the skeleton for the primitive, mainly consisting of its control flow, is given by the JDC's black-box specification: every possible method call and data-usage must appear in the black-box specification. In particular:
  - the service methods are obtained from the behaviour associated with the same method name in the JDC. We rely on the strong functional behaviour encapsulation of GCM for this matter;
  - ProActive's `runActivity()` method, i.e. the service policy of the active object implementing the primitive is directly generated from JDC's serving policy;
- then, the programmer must implement the code that deals with data management details, but should not modify the control code, so that the code is certified to comply with the specification.

16.4.3 Generation of Behavioural Models

Again, the generation of behavioural models from JDC specifications is not implemented yet, but we state that JDC specfications are suitable for such. The

checking tools used in our verification platform [6] are based mainly on explicit-state models, so we need to build models that abstract the concrete behaviour of our components in a way that preserves our specifications. We plan to do this in three steps. First, we shall transform the original JDC specification using the mapping to simple types. Then we shall build pNets models as described in previous work [9, 10]. Finally, for each formula to be checked, we will use finite partitions as a domain abstraction for each parameter in the model, that will give a finite model usable in a model-checker. In [6], we described a tool called ADL2N for quasi-automatic behavioural model generation. In JDC a similar technique applies, but with a better expressivity as JDC includes information about control and data flow.

We only comment here the pNets construction. It involves two aspects: mapping (abstracting) user data types, i.e., mapping arbitrary Java classes into Simple Types that are the only parameter domains allowed in pNets as described in Section 16.2.3; and building synchronisation objects corresponding to the various architecture and communication primitives of JDC.

Building the behaviour model for a JDC component requires to: (1) turn the pseudo code in each JDC method (the service policy and service methods) into a pLTS (parameterized Labelled Transition System) (2) generate pLTSs for specific primitives or library element of JDC, e.g. request queues, NF controllers, etc; (3) generate synchronisation structures (pNets) for relating these pLTS, depending on the types of bindings and interfaces used.

16.5 Analysis

The models obtained in the previous section allow us to generate both parameterized and finite abstractions of the system behaviour, either for a single component, or for an arbitrary assembly. In principle, this allows for checking:

- 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 formulas, 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 architecture) and its black-box specification, expressed as an equivalence or a preorder relation (as in [8]).

In this chapter we only consider verification performed on the finite abstraction of the model, and the verification is done by the Evaluator model-checker (from the CADP toolset). Both the parameterized case (using "infinite-state" engines), and the conformance checking are left for further work.

In the following pages, we give examples of verification of the CoCoME requirement scenarios, that we have performed with the current version of CTTool, and explain a number of results of this analysis.

16.5.1 System Verification

The JDC tool support itself being not yet available, we have conducted the analysis activities in this section using directly the capabilities of CTTool: it works by generating LOTOS code that implements a (synchronous) semantics of UML component diagrams and state-machines (including data-types), and passing this LOTOS code to the CADP verification toolset. CTTool itself includes a user interface that hides most of the verification engine complexity, and provides a number of menus for controlling the CADP functions.

There are many ways of encoding formulas. Some of them are very powerful as μ-calculus, but at the same time hardly usable by non-experts. So, having software engineer's expertise and JDC in mind, we propose to write formulas using automata. Transitions contain predicates with logic quantifiers and states can be marked as either acceptance or rejection. The automata may change to any state whose transition predicates are satisfied. If a final state is unreachable, the formula is false. Moreover, there are special predicates:

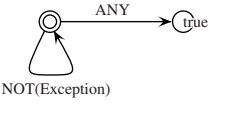
- `ANY` meaning that any label satisfies the predicate;
- `NOT(i), (i AND j), (i OR j)` meaning that any label but `i` given satisfies the expression, both `i` and `j` must satisfy the expression and either `i` or `j` may satisfy the expression respectively;
- and `ANYOTHER` meaning that all labels not satisfying other transitions from the state satisfies the predicate.

Then, formulas are readable and easy to write, and have language constructs compatible with both CTTool's state-machines and JDC's regular expressions.

Absence of Deadlocks. There are basic formulas that can be proved, the most common being the absence of deadlocks. In the case of our CTTool specification, this ends-up being trivially false because of two reasons:

- the specification of exceptions: in our specification, any time an exception is raised, we just block the system. So we may want to search for deadlocks that are not following an exception;
- the synchronous semantics of CTTool components: in CTTool, components are mono-threaded, and communications are synchronous. As a result, the system deadlocks due to race conditions over the EventBus. Concretely, events are not atomic within the EventBus, so a controller may trigger an event (and therefore block the EventBus) while the Application is running an ongoing sale. At this moment, if-ever the Application needs to access any of its controllers (through the EventBus), the system deadlocks.

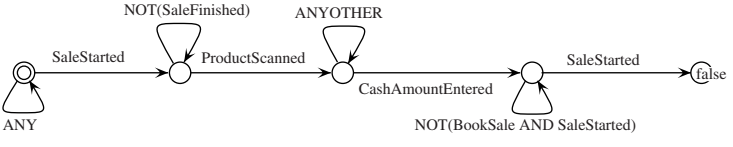
To show this, we write a formula expressing that all deadlocks are the consequence of an exception, and to model-check this formula. More precisely we write the negation, i.e. that any transition is followed by some other transition as long as an exception has not been raised.



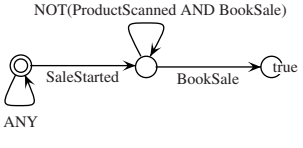
The answer we get when we evaluate the former formula is "false" (the formula does not stand). As a diagnostic we receive a trace in which the ScannerController triggers a ProductBarCodeScanEvent, blocking the EventBus. Meanwhile, the application is trying to synchronise with the EventBus for querying the price of the previously scanned product.

Note that these kind of scenarios would not be present in a real ProActive application because of the asynchronous method calls which buffers requests in the queues: we have more deadlocks in a synchronous implementation of the system than those we would have with ProActive. Nevertheless, using the CTTool specification we were able to prove some interesting scenarios, and to find some errors (or underspecifications) within the reference CoCoME specification.

Main Sale Process. This scenario relates to "Use Case 1" (Chapter 3, Figures 15 and Figure 16) To illustrate the capabilities of our approach to verify more specific properties, we have checked (still in the synchronous model) some of the usage scenarios listed in the CoCoME specification. Our first scenario is defined in CoCoME's requirements as a trace in UML sequence diagram. We successfully verified that this trace is feasible in the state-space generated by CADP. An even more interesting scenario can be encoded as a negation of the following: a sale starts; before it finishes, valid products are scanned; the client pays with enough cash; the sale is not registered and a new sale starts. CADP's diagnostic strikes out that it is not possible to start a new sale before booking the former one.



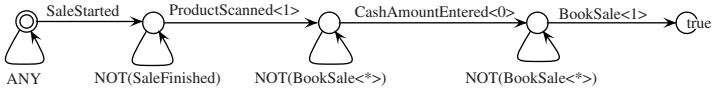
Booking an Empty Sale. This scenario relates to "Use Case 1" (Chapter 3, Figure 16)Although it may not be an error, it is strange that a system allows an empty sale to be booked. This trace was found when searching for the shortest path (by using Breadth-First Search algorithm) that books a sale. Alternatively, a software engineer may want to explicitly verify if this scenario is feasible with the automaton below. A sale is started; no products are scanned; the sale is booked.



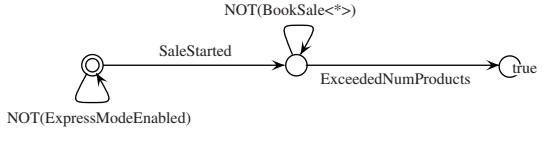
Successful Sale with Insufficient Money. This scenario relates to "Use Case 1" (Chapter 3, Figures 15 and Figure 16) We found that it was possible to book a sale even if the client pays with insufficient money. The problem is reflected by the fact that there is no way of aborting a sale when paying with cash, and

there is no verification whether the money fulfils expenses. Note that this issue was verified running the CoCoME reference implementation – which allowed to pay \$0 – and going through the UML specification – which has nothing relative to it. In fact, the insufficient funds exception was only foreseen within a credit card payment. Nonetheless, in general, once a sale is started, it is not possible to abort, so the system must book the sale before proceding.

Note that for this scenario, the data abstraction plays an important role as it plays a role in the control-flow of the application. The amount that the client pays then is abstracted with two values: one with insufficient money and the other with sufficient money.



Safety of the Express Mode. This scenario relates to "Use Case 2" (Chapter 3, Figure 18). A non-precised scenario was found. There is nothing within the CoCoME reference specification that states when a CashDesk may switch from/to an express mode. In fact, the system ends-up in an inconsistent state if an express mode signal is triggered during an ongoing sale. This scenario can be found using the following property:



16.6 Summary

In this chapter we presented the Grid Component Model (GCM) and the Java Distributed Component specification language (JDC) as a means to address robust distributed component system design. Our approach aims at integrating at the level of the specification language the architecture specification and its implementation together with the black-box behaviour of the components in order to generate safe code by construction.

We started by presenting the GCM and its reference implementation using the ProActive library. Then, we have defined the JDC specification language as the kernel of our framework. Then, we gave the JDC specification of two of the CoCoME components, the full specification being available in an extended version of this chapter.[2] In addition we gave an alternative method for building JDC specifications, using UML 2 diagrams available in our CTTool software. From these specifications, we created by hand a GCM/ProActive implementation showing how the control code can be automatically generated from the JDC

[2] `http://www-sop.inria.fr/oasis/Vercors/Cocome/`

specification. From the same specification, completed by abstraction functions, we have shown how to generate models suitable for verification, in the form of parameterized networks of synchronised transition systems.

Finally, the CTTool specification was used to check for safety properties of the reference CoCoME specification scenarios. Because of our synchronous encoding, the EventBus component ended-up being a source of deadlocks. As a result of this verification activity, we found a number of interesting features, that we interpreted as bugs or under-specifications in the CoCoME official definition.

Limitations. There are number of features that are not considered in this approach: we do not consider any "performance" aspect of the specification (response time or other quality of service measures); neither do we try to fully specify the functional part of the code (data computation), as could be done in some proof-assistant based approaches.

Other limitations come from the early state of some of our software platform: our model generator is currently limited to the synchronous interpretation of the component systems (only the synchronous controllers are generated), so the features relying on asynchronous communication, and in particular the existence of functional deadlocks in Grid-based systems cannot be analysed. These developments are planned for the next version of the tool. Currently we have no direct support for the JDC language itself, so we rely on the existing CTTool platform to provide an alternative tool suite. We are working on an analysis platform with direct support for JDC, in which the CTTool diagrams will only be an alternative syntax for the specification. The JDC development platform will include an Eclipse plug-in, a Java code generation tool, a model generation tool, and a JDC formula compiler. With them, we hope to answer the software engineer's needs when developping distributed components.

References

1. 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)
2. Caromel, D., Henrio, L.: A Theory of Distributed Object. Springer, Heidelberg (2005)
3. Bruneton, E., Coupaye, T., Leclercp, M., Quema, V., Stefani, J.: 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. Springer, Heidelberg (2004)
4. OMG: Corba components, version 3. Document formal/02-06-65 (2002)
5. Cansado, A., Henrio, L., Madelaine, E.: Towards real case component model-checking. In: 5th Fractal Workshop, Nantes, France (July 2006)
6. Barros, T., Cansado, A., Madelaine, E., Rivera, M.: Model checking distributed components: The Vercors platform. In: 3rd workshop on Formal Aspects of Component Systems. ENTCS, Prague, Czech Republic (2006)
7. Jezek, P., Kofron, J., Plasil, F.: Model checking of component behavior specification: A real life experience. In: International Workshop on Formal Aspects of Component Software (FACS 2005). Electronic Notes in Theoretical Computer Science (ENTCS), Macao (2005)

8. Černá, I., Vařeková, P., Zimmerova, B.: Component substitutability via equivalencies of component-interaction automata. In: Proceedings of the Workshop on Formal Aspects of Component Software (FACS 2006). ENTCS, Prague, Czech Republic (to appear, 2006)
9. Barros, T., Henrio, L., Madelaine, E.: Behavioural models for hierarchical components. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639. Springer, Heidelberg (2005)
10. Barros, T., Henrio, L., Madelaine, E.: Verification of distributed hierarchical components. In: International Workshop on Formal Aspects of Component Software (FACS 2005). ENTCS, Macao (2005)
11. Barros, T., Boulifa, R., Madelaine, E.: Parameterized models for distributed java objects. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235. Springer, Heidelberg (2004)
12. CoreGRID, Programming Model Institute: Basic features of the grid component model (assessed), Deliverable D.PM.04 (2006), `http://www.coregrid.net/mambo/images/stories/Deliverables/d.pm.04.pdf`
13. Caromel, D., Henrio, L., Serpette, B.: Asynchronous and deterministic objects. In: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Venice, Italy, pp. 123–134. ACM Press, New York (2004)
14. Apvrille, L.: Turtle documentation (2005), `http://labsoc.comelec.enst.fr/turtle/help/`
15. Garavel, H., Lang, F., Mateescu, R.: An overview of CADP 2001. European Association for Software Science and Technology (EASST) Newsletter 4, 13–24 (2002)