Unifying Architectural and Behavioural Specifications of Distributed Components

Antonio Cansado, Ludovic Henrio, Eric Madelaine

Unifying Architectural and Behavioural Specifications of Distributed Components

Antonio Cansado, Ludovic Henrio, Eric Madelaine

INRIA Sophia Antipolis, CNRS - I3S - Univ. Nice Sophia Antipolis
2004, Route des Lucioles, BP 93, F-06902 Sophia-Antipolis Cedex - France
Email :First.Last@sophia.inria.fr

Abstract

We present a novel specification language called JDC to be used at design phase of distributed components. The extensive seek for asynchrony in distributed components demands new techniques for its specification that have not been addressed before. We propose to focus the specification on its data-flow; this allows to reason about inter-component synchronisations produced by a data-driven synchronisation model. The language is endowed with enough formality so it allows a constructive approach; it allows the generation of behaviour models which can be model-checked, and the generation of code skeletons with the control flow of components. Globally, this approach aims at generating components with strong guarantees w.r.t. their behaviour.

Keywords: Hierarchical components, distributed asynchronous components, formal verification, behavioural specification, model-checking, specification language.

1 Introduction

Component-based software development (CBSD) has emerged as a response from both the industry and the academy for dealing with software complexity and reusability. The main idea is to clearly define interfaces between components so that they can be assembled and composed in several contexts. Unfortunately, software engineers often face non-trivial runtime incompatibilities when assembling off-the-shelf components. These arise due to an inadequate (or nonexistent) dynamic specification of the component behaviour. In fact, only few state-of-the-art implementations of component models take into account dynamic compatibility. The component models SOFA [17] and Fractal [5] can be specified using "behavior protocols" [17], or (for Fractal) with our pNets formalism [2]. Other component models such as CORBA Component Model [16] only check interface type-compatibility in order to realise a binding. Types are defined in an Interface Description Language (IDL).

A major originality of our work is that we target distributed component systems communicating by asynchronous method calls with futures, concretely in the frame of the Grid Component Model (GCM) [12]. The GCM is a novel component model

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

defined by the european Network of Excellence CoreGrid. The GCM is based on the Fractal Component Model, with extensions addressing Grid computing. From Fractal, GCM inherits a hierarchical structure with strong separation of concerns between functional and non-functional behaviours. The extensions to Fractal come from the fact that in Grid computing components are deployed over thousands of nodes, so scalability plays a major role.

Even if there are many specification languages in the literature, none fits well in the context of distributed components. In the GCM, most difficulties come when specifying the synchronisations. From a practical point of view, we focus on a reference implementation of GCM in Java: GCM/ProActive. In GCM/ProActive [8], components communicate through asynchronous method calls with futures. Futures act as placeholders for promised return values. Synchronisations happen upon data access on a future, and futures can be transmitted in remote method calls to other components; finally, almost any object in the program can be a future or not in a transparent way. Such transparent futures alleviate the programmer from synchronisation difficulties, allow for separation of concerns (the source code can be really independent from the physical infrastructure), and give optimisation opportunities at the middleware level. On the other hand, specifying and/or inferring about synchronisations becomes more complex; we need to provide help to the programmer. To our knowledge, no specification language has been proposed within this context.

Our approach in [7] was to attach the behaviour of components as part of the architecture specification, defined in terms of *Parameterized Networks of Transition Systems* (pNets) [2]; pNets is a powerful model that expresses parameterized topologies of processes communicating with value passing. Using pNets, we showed how to synthesise the behaviour of distributed components; however the formalism is too low-level to be used as a specification language, and lacks of the high-level concepts particular to the different contexts in which we want to use it.

Related work: In the same spirit, "behavior protocols" [17] is an ongoing research project that seeks formal specifications of components. They opt for simplicity rather than expressivity, for example "behavior protocols" uses a simple regular-language to describe traces of the component behaviour. This allows them to check for behavioural mismatches, however they only take into account a limited abstraction of the data-flow.

STSLib [15] provides a formal component framework that synthesises components from symbolic protocols in terms of Symbolic Transition Systems (STS). Just as pNets, STS concisely represents infinite systems, however, STS relies on Abstract Data Types (ADT) which are more expressive than our Simple Types (see Section 2.3), but less intuitive for software engineers. Both formalisms rely on (N-ary) synchronisation vectors, but in STS they are static whereas in pNets they are dynamic; as shown in [2], this allows us to express reconfiguration in a natural way: rebinding a set of interfaces is seen as a change in the synchronisation vectors. STSLib synthesises components based on their STS protocols; a controller interprets the STS protocol and data from the ADT is implemented (and generated) in Java. The communication in STS components is rather low-level; both emitter and receiver must agree exchange a message, although there is no clear notion of required nor provided services.

Sensoria [1] is another project which provides a mathematical framework for component interaction. It targets Service Oriented Architectures (SOA) such as Web Services and SCA (Service Component Architecture [4]). Their approach is akin with "behavior protocols", specifying the allowed interaction within the system. Our approach is closer to the programming model, expressing *what* the component does to later infer *which* are the interactions.

Contribution: The originality of our work is to focus on service invocations, and implicit synchronisation by the mean of futures. We will show that the data-flow and the access to the transmitted results implicitly set the synchronisations. This approach provides a high-level and powerful abstraction for the programmer that is close to the programming model.

Instead of proving that legacy code is safe, in this paper we take a constructive approach similar to [11,15]. The idea is to specify the system, prove that the specification is correct, and then generate (Java) code skeletons guaranteed to conform to the specification. pNets is left as the underlying formalism that interfaces with model-checkers, and the programmer uses a high-level specification on top of pNets. The language is called *Java Distributed Components* (JDC for short).

Paper structure: The paper is organised as follows. Section 2 discusses the foundations of the specification language. Then, Section 3 illustrates how components can be described and composed using an architecture specification. In Section 4, we define the black-box behaviour of a component, that abstracts the internal details of a component. Section 5 specifies abstractions of user types. Finally, Section 6 explains how to generate both behavioural models and code skeletons from our specification language.

2 Foundations of the Specification Language

Distributed components tend to be coarse grain units of composition, and are often loosely-coupled. In the following we present a specification language in the form of an extension of a subset of Java for specifying these components. The language includes both the architecture and the behaviour definitions, and is endowed with enough formality and control-flow information so that we are able to:

- on one hand check the correctness of the system (Section 6.2): we build a behaviour model that can be model-checked against temporal formulas;
- on the other hand generate safe components (Section 6.3): we generate the control code of components that is guaranteed to respect the specification.

We opt for a Java-like language for several reasons; (i) it is close to the target expertise of engineers, using common syntax such as method calls and data classes; (ii) it allows to embed part of the specification within the code skeletons; (iii) it uses the same datatypes as in the implementation, guaranteeing that operations on the datatypes are directly useful without modification.

2.1 Background on Distributed Components

A recent approach to deal with distributed components on Grids is provided by ProActive [8], the reference implementation of the GCM. Components communicate

through asynchronous method calls. A method call creates a request in the queue of the target component, and a future on the caller side as a placeholder for the result. These futures may be transmitted between components, no explicit instruction deals with futures, neither for creation nor for access, but access to the queue is explicit. `serve(`*method*`)` is used to select methods from the queue.

ProActive guarantees that, once the promised value of a future is known, it is transmitted to every component that has received a reference to it. Moreover, the various strategies used for transmitting the future values are proved not to change the component behaviour. A precise operational semantics of ProActive is given by the ASP-calculus [9]. These results inspire our specification language to adopt futures in order to decouple components.

Using transparent futures in the specification language brings the same advantages as in the programming language: the system designer doesn't have to wonder if a variable might contain a future; or more precisely, no explicit synchronisation mechanism is needed for variables that may sometimes contain a future. This extends reusability of specifications as they may fit several contexts, where values are remotely computed, or come from local instances. A drawback of transparency of futures is non-determinism; it is in general not statically decidable whether a variable is a future or not at a given point of the program. However, additional synchronisation can be specified, ensuring that, after synchronisation upon a variable, this variable is known to be value, or a future with a filled value.

Dynamic reconfiguration is supported in the GCM, however, it is not yet considered in JDC. In [2] we proposed models, based on our pNets, to handle reconfiguration of components. We plan to extend the language towards this direction.

2.2 Decomposing the Behaviour into Services

The functional behaviour of the component is an abstraction of the control-flow, some elements of data-flow, and access to data. Concretely, for the distributed components we deal with, the interesting events are:

- *Remote method calls*, these represent communication between components. A remote call is always an asynchronous, it creates a request in request queue of the callee component, and it creates a future in the caller for dealing with the promised result. Remote calls are identified by calls on client interfaces.
- *Future flow*, these represent the creation of implicit communication channels between the component that computes the value of the future, and the component that receives the reference to the future. The future flow can be identified by tracking future objects in parameters and results of remote method calls.
- *Data-access*, these trigger synchronisations between components. They are identified using static analysis, or given explicitly within the specification.

The first part of the component specification is called the *service policy*; it defines how a component selects requests depending on its internal state, and any behaviour the component triggers by its own. This is a rough specification of the component protocol, however, it gives the user a good idea of how the component should be used. For instance, the specification may specify that a component must

serve requests in a particular order.

The second part of a service specifies what each request exposed at the service policy actually does. This behaviour is defined by a Java-like language that is very close to the programming model we want to specify. In there we include an abstraction of the control and data flow, remote method calls done within the service method, and access to data. Although it requires static analysis to infer the behaviour, it is easier than in standard Java; remote calls are easily identified by calls on the component's client interfaces; future creation points are identified as the results of these calls; there is no concurrency within the service method; and there is no exception handling (for the sake of asynchrony).

2.3 Datatypes and Abstraction

The datatypes used in JDC are standard Java classes. This way the code-skeletons obtained by our generation tools will be directly usable. On the other hand, arbitrary datatypes often have large (possibly infinite) domains which can't be model-checked directly. The kind of behavioural properties we seek only require an abstraction of these datatypes. Therefore, whenever verification is desired, the specification includes as well an abstraction of the user types that allows to derive a simpler specification.

The abstraction keeps solely data influencing the control-flow and the synchronisations, however, it must preserve the behavioural properties in the sense of Cousot's abstract interpretations [13]. If abstractions are finite and constitute abstract interpretations of the initial parameter domains, then the model is finite. Following [10], we build an abstract interpretation of the system behaviour, from abstractions of the domains of the program variables; this construction can be used for finite model-checking as it preserves safety and liveness properties.

The abstractions are mappings from user types to predefined first order datatypes (*simple types* for now on). Simple types themselves are provided as Java classes, and as a particular case, can be used in JDC programs. They are: point (or singleton), booleans, enumerated types, integers, intervals of integers, strings, records of simple types, arrays of simple types.

In our work we decompose the abstraction in two steps: the first maps concrete types to potentially infinite *simple types* allowing us to generate parameterized *pNets* models. From pNets, we can apply many different proof methods, including inductive theorem proving techniques, that can address a large family of properties. The second step is based on finite partitions of parameter domains that depend on each set of properties to prove. In this case, the abstraction produces finite pNets on which we can use explicit-state model-checkers.

Finally, our abstractions must consider futures. Even if a variable has insignificant values, access to the variable may still trigger synchronisation. This makes the choice of a good abstraction tricky, and some variables are only kept within the abstraction in order to signal eventual access on them. In other words, these variables have an abstract domain with 2 values *filled* or *non-filled*.

3 Architecture Specification

In the next sections, we present elements of the abstract and concrete syntax of JDC. Each box defines a piece of JDC syntax, using: keywords in bold (e.g. **component**); terminal symbols written between simple quotes (e.g. '{'); non-terminal symbols in monospace (e.g. `Services`); optional expressions with square-brackets (e.g. [`expr`]); choices with | (e.g. `expr1` | `expr2`); concatenations of zero (resp. one) or more expressions with * and + (e.g. `expr`*, `expr`+); and identifiers: 'id' .

3.1 Defining a Component

The definition of a component type comprises its external interfaces with both provisions and requirements, and a specification of its behaviour. The behaviour is either given by a black-box specification in the form of a set of *Services* (Section 4), or by a composition of components, also called *Architecture* (Section 3.2), or even by both.

`Component` →	**component** 'id' '{'	≪component definition≫
	external interfaces	
	`Interface`*	≪set of interfaces≫
	[`Services`]	≪black-box description≫
	[`Architecture`]	≪content description≫
	'}'	
`Interface` →	**server** | **client**	≪interface role≫
	interface `InterfaceType` 'id' ';'	≪type and name≫

Each interface in a component has a role (either server or client), a type (a Java interface as in most IDLs), and a name. The interfaces defined within the context of the component definition are *external interfaces* and can be bound to the environment. Interfaces determine both provided and required services of a component; provided services are defined by server interfaces, and required services are defined by client interfaces.

3.2 Composing Components

The composition of components is done within the *architecture*. It exposes the content of a component by means of its subcomponents, its internal interfaces, and the bindings. The subcomponents are named and typed, the type being given by either an external component definition, or by an inline definition. The bindings connect two interfaces among the component's internal interfaces and the subcomponents' external interfaces.

In the GCM, the relation between an internal interface and an external interface of a component is arbitrary: *interceptors* can transform or intercept any incoming invocation. For simplicity, in this paper, we assume that there is an exact match for each pair of external-internal interfaces (interfaces that have the same type and name, but with opposite roles); and that invocations on an external (resp. internal) server interface is directly forwarded to the corresponding internal (resp.external) client interface.

```
Architecture →  architecture
                  contents
                    Subcomponent*                    ≪set of subcomponents≫
                  internal interfaces
                    Interface*                       ≪set of interfaces≫
                  bindings
                    Binding*                         ≪set of bindings≫
Subcomponent →  ComponentType 'id' ';'               ≪named subcomponent≫
                | Component                          ≪inline definition≫
ComponentType → 'id'                                 ≪reference to a type≫
      Binding → bind '(' SourceItf ',' TargetItf ')' ';'   ≪binds a pair of interfaces≫
```

3.3 *Example*

The CoCoME example [6] was implemented using GCM / ProActive. It is a Point-Of-Sale system, in which the cash desk deals with the sales. The Cash Desk and its hardware controllers are implemented as components, depicted in Figs. 1 and 2.

```
component CashDesk {
  external interfaces
    server interface ApplicationIf appIf;
    client interface ScannerIf scannerIf;
    // ... external interfaces
  architecture
    contents
      component Application application;
      component Scanner scanner;
      // ... controllers
    internal interfaces
       server interface ApplicationIf appIf;
       // ... internal interfaces
    bindings
      bind(this.appIf, application.appIf);
      // ... bindings
}
```

Fig. 1. *Architecture* specification

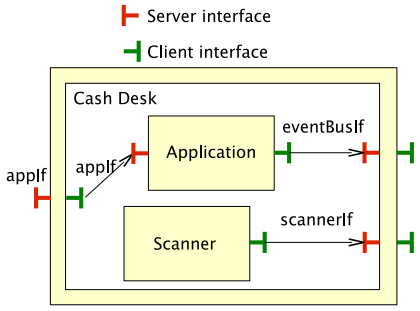

Fig. 2. Equivalent schema

4 Behaviour Specification

When designing a system, the designer would like to adopt a top-bottom approach: specifying first the behaviour of a component before going down into its architecture. Thus, we also propose to specify directly the behaviour acceptable by the interfaces; this is called a *black-box* behaviour of a component. Of course different *architecture* definitions can match the same component *black-box*. In this paper, we leave the equivalence (or preorder) between a component *black-box*, and its implementation (*architecture*) unspecified. Many existing work can apply, 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].

In GCM there are two kinds of components, *primitives* that are atomic components, and *composites* that are components composed of other components. Primitives are monothreaded, and concurrency is introduced by composites. The concurrency in JDC is specified by a set of concurrent services within the *Services* block.

Each *service* denotes a sequential process with its own set of local variables. A sequential process is split into the *service policy* that defines the high-level protocol of the service, and a set of *service methods* that details the behaviour of the methods exported by the component.

```
Services →  services
              Service+                  ≪one or more concurrent services≫
Service →   service  '{'
              LocalVariableDecl*        ≪variables of the component≫
              policy  '{' Policy '}'    ≪service policy≫
              ServiceMethodDecl*        ≪service methods (exported)≫
              LocalMethodDecl*          ≪local methods (not exported)≫
            '}'
```

4.1 Service Policy

The service policy defines how incoming requests are selected from the queue depending on the internal state of the component, and any behaviour triggered internally. It is given by (non-deterministic) state-machines, expressed using regular expressions. The actions can express *reactive* or *active* behaviour.

```
Policy →     ServeMode '(' [ Filter ] ')'     ≪reactive service≫
           | MethodCall                       ≪active service≫
           | Policy ';' Policy                ≪concatenation≫
           | Policy '|' Policy                ≪choice≫
           | Policy '*'                       ≪Kleene closure≫
ServeMode →  serveOldest | serveYoungest      ≪request queue accessor≫
Filter →   | InterfaceName                    ≪any method in this interface≫
           | InterfaceName '.' MethodName     ≪this method≫
           | Filter ',' Filter                ≪a list of filters≫
```

The *reactive* behaviour defines which kind of methods to select, and in which order to pick them from the queue. This represents work that depends on the requests at the component's request queue. As an example, `serveOldest(itf.m1, itf.m2)` selects from the queue the oldest request on `m1` or `m2`; if none of them is in the queue, the service blocks until one of them arrives. Then, the request is served, i.e., the control is delegated to the service method representing the request.

Additionally, an *active* behaviour denotes spontaneous behaviour, i.e., some work that is done without being requested. In our example, a component in charge of the scanner sends signals to the application component whenever a product is scanned. The signals take the form of method calls on the application components. For the scanner component, this behaviour is spontaneous as the interaction with the physical scanner is abstracted away.

The service policy is the only block authorised to access the queue. Basically, this ensures that the code generated for the service policy will be complete w.r.t. how the component provides services. Moreover, the state-machines are precise enough to ensure that the code generated will be the final implementation of the

`runActivity()` method of a GCM/ProActive component, and no other method will access the component's request queue. More details are discussed in Section 6.3.

An example of a *Service* definition is found in Fig. 3. We give part of the behaviour of the cash desk application. It has a single service (the component is indeed monothreaded), and it is mainly reactive; it responds to incoming events in a FIFO order.

4.2 Concurrent Behaviour

A primitive component can be specified by a single *Service*. This specification fits as well in a composite component with a pipeline of subcomponents inside. In any of these configurations, two request calls are treated sequentially. However, a single *Service* cannot express concurrency as there is no explicit thread creation in JDC. Instead, concurrency of requests is defined by multiple services within a component. Each service is an independent activity serving requests in parallel, with its own set of local variables and provided services.

A drawback of this approach is that it is not possible to define interference among the services directly. That is, we must rely on an *architecture* definition that composes independent components in order to express interference. Other alternatives would have introduced more complexity to the language; moreover, the generation of the control code would have been difficult as the programming model doesn't have explicit concurrency.

```
services
  service {
    // variables of simple types
    Bool expressMode;
    public enum CashState{
       IDLE, STARTED, PAYING
    }
    CashState cashState;
    // ... other variables of
         simple types and user types

    // initialises the system with
         some RPC and then treats
         calls in FIFO order
    policy {
       init(); // local method
       serveOldest(applicationIf)*
    }
   // ... the service methods
 }
```

Fig. 3. Service definition of the Cash Desk Application

```
void applicationIf.barcodeScanned(Barcode barcode) {
  switch (cashState) {
    case IDLE:
    case PAYING:
      break;  // ignore signal
    case STARTED:
      Product product = cashDeskIf.getProduct(barcode
          );
      if (product == null) {
        eventBusIf.productBarcodeNotValid();
        break;
      }
      if (expressMode && products.isFull())
        __ERROR("ExceededNumberOfProducts");
      else {
        products.add(product);
        runningTotal.add(product.getPurchasePrice());
        eventBusIf.runningTotalChanged(runningTotal,
            product);
} }   }
```

Fig. 4. A Service Method of the Cash Desk Application

4.3 Service Methods

A service method is an abstraction of a service exported by a component. It is defined by means of a subset of Java statements in which there is no exception handling, and no concurrency. This includes the relevant dataflow between input parameters and results of the method, as well as communication with required services. The service method has access to the component's variables, however, it doesn't access the component's request queue.

Java is extended to deal with component interfaces. The name of the service method is prefixed by the server interface in which it is defined. Client interfaces are accessed as usual objects but they *cannot be assigned to other variables*. This last requirement is very important to ensure that all the interactions between components are realised through the client interfaces.

An example of a service method is depicted in Fig. 4. The behaviour focuses on a cash desk that may provide an express mode for dealing with sales with a limited amount of products. When the barcode of a product is scanned, the component reacts accordingly to its internal state. Its usual behaviour is to get the product information by invoking a remote method call (`getProduct(barcode)`), add the product to a list of `products`, and update some information regarding the current sale (`runningTotal`). The specification is quite close to Java, notably the operations on the product are the ones that would be expected in a real implementation.

5 Specifying Abstractions

This section shows how to define and use abstractions of user types in JDC. One particularity is that a class may have more than one abstraction defined, each one focusing on the significant behaviour of a variable.

The abstractions ensure that we are able to generate behavioural models based on pNets. pNets allows us to interface with several verification tools; for the moment we focus on finite-state model-checkers, but using pNets we can potentially interface with infinite-state model-checkers and theorem provers as well.

5.1 Formalisation of an Abstraction

A class is a tuple $\mathcal{C} =< \overrightarrow{m}, \overrightarrow{f} >$, where $\overrightarrow{m} = \{m^i(\overrightarrow{a}) : \tau^i\}$ are the methods of $\mathcal{C}$; $\overrightarrow{a} = \{a^j : \tau^j\}$ are the method arguments; and $\overrightarrow{f} = \{f^k : \tau^k\}$ the fields.

An abstraction of $\mathcal{C}$ is a class $\mathcal{C}_\mathcal{A} =< \overrightarrow{m_\mathcal{A}}, \overrightarrow{f_\mathcal{A}} >$, where each public method $m(\{a^j : \tau^j\} : \tau)$ of $\mathcal{C}$ has one or more abstract method $m_\mathcal{A}(\overrightarrow{a_\mathcal{A}}) : \{\tau_\mathcal{A}\}$ with $\overrightarrow{a_\mathcal{A}}$ the abstract arguments, which domains are sets of values in the abstractions of classes τ^i, and the result is an abstract value in the abstraction of class τ.

For defining what is a good abstraction of the domains of the variables in the specification, we need to identify:

- where in the specification are the "variables of interest" – those used in the properties to be proved;
- what are the significant values of these "variables of interest" – these will determine their abstract domain;
- which other variables in the program influence (through control-flow and data-flow) the "variables of interest" – these other variables will also have a non-empty abstract domain.

For each of these significant variables, we must attach an abstract type in the following manner:

- for each public method m of $\mathcal{C}$, abstract versions $m_\mathcal{A}$ are provided that capture the accesses on the class variables, accesses on the variables passed as arguments,

and relevant results of them.

- the fields of the concrete class that are of interest are included as a record. The domains of these fields are such that they are precise enough to hold the property to prove. This is done recursively in order to find the abstractions of the other variables of interest.

5.2 Using Abstractions

An abstraction in JDC is similar to a Java class, with extensions to deal with non-determinism and data abstraction. An important notion is that we may have to use different abstractions for different variables of the same concrete type, within a given program. This means that in the abstract program, we may need different versions of the abstract operators, depending on the abstract types of the arguments. For example, if the concrete program has variables `x:Int, y:Int` then the abstract program may have `x:Sign, y:[0..3]`, and we may need to define the `+` operator for arguments in `Sign and [0..3]`. We solve this problem in two phases: we define a library of abstract classes (here `Sign` and `interval` as abstractions of `Int`, with the standard abstract operators in each (e.g. `+ : Sign*Sign -> Sign`); these libraries can be defined in a generic way, and reused easily. Then for a specific program, we define abstract classes that inherit from the required library abstract classes, and define additional abstract operators depending on the specific abstraction of variables, and of the occurrences of the operators found in the code (e.g. `+ : Sign*[0..3] -> Sign`).

`Abstraction` →	**abstraction** 'id' **of** 'id' '{'	≪datatype abstraction≫
	`Field`*	≪local variables≫
	`Constructor`*	≪abstract constructors≫
	`Operator`*	≪abstract operators≫
	'}'	
`Field` →	`Type` 'id'	≪type and name of variable≫
	[**abstracted as** `Type`]	≪local mapping of a type≫
`Operator` →	`Type` 'id' '(' `args` ')'	≪signature of concrete operator≫
	[**abstracted as** `Type` 'id' '(' `args` ')']	≪signature of abstract version≫

The fields within an abstraction are variables of type *simple type*, or any other usertype provided with an abstraction. The latter can be given by a unique global abstraction for the type, or by an inline abstraction that selectively determines the abstraction for the type.

The operators are abstract versions of the class methods, that capture the behaviour of interest for a variable. It is possible to have multiple versions of the same operator, each one taking different abstract versions of the arguments and return types. Similarly, the same applies to constructors.

It is often useful (or required) to underspecify what are the results of an expression, possibly as the result is a set of abstract values. The language includes for that two non-deterministic operators; the first, called `ANY`, non-deterministically returns any element of the abstract domain; the second, called `ANYELEMENT`, non-deterministically selects an element from a list.

Moreover, it is often not possible to statically know if a variable refers to a value or to a future. The safe assumption is to consider such variable as *possibly future*. In here, we exploit that a *non-future* variable is semantically equivalent to a *future* variable with filled value. Nevertheless, the user must keep in mind that some traces in the specification may never occur in a concrete implementation. A solution can be then to make the specification more precise by enforcing more synchronisation on a variable (by means of `touch()`). After the synchronisation, the variable is known to be *non-future*. `touch()` synchronises on the variable without describing which operations are applied. This allows details of the implementation to be filled-in later without changing the synchronisations occurring in the system.

5.3 Example

```
abstraction ListProducts_A of ListProducts {
   enum ListState { EMPTY, OK, FULL }
   List<Product> products abstracted as
        ListState;
   ListProducts() abstracted as ListProducts_A
        () {
      products = EMPTY;
   }
   Bool isFull() { return (products==FULL); }
   Product get() abstracted as Product_A get()
         {
      switch(products) {
         case EMPTY:
            return null;
         case OK:
            if (Bool.ANY())
               products = EMPTY;
            return Product_A.ANY();
         case FULL:
            products = OK;
            return Product_A.ANY();
    } }
   void add(Product product) abstracted as
        void add(Product_A product){
      product.touch();
      switch(products) {
         case EMPTY:
            products = OK;
            break;
         case OK:
            if (Bool.ANY())
               products = FULL;
            break;
         case FULL:
            break;
} } }
```

The example above illustrates the use of a data abstraction influencing the control-flow. A short-sale must not exceed a maximum number of products, but there is no constraint on the type of products. Therefore, the abstraction of the product list must be precise enough to take into account whether the maximum has been exceeded or not, and can abstract away the product information.

The abstraction for the product list has no counter. Instead, it focuses on the states the list can have: the list is either `EMPTY`, `OK` or `FULL`. This abstraction is imprecise w.r.t. the number of products it has, so actions on the list are non-deterministic. Adding a product from an `EMPTY` state never reaches the limit for a short-sale, however, from an `OK` state it may (the state change to `FULL` is non-deterministic). Note that the context guarantees that we never call `add()` when the list is `FULL`.

The abstraction for the `product` is such that we are able to signal access upon the variable. This is necessary as the `product` may be a future; indeed, in Fig. 4 `product` is the return of a remote method call and thus can be a future. Therefore, the `product` is abstracted as a Singleton domain (`Product_A`) such that the access is signalled by `touch`.

6 Work in Progress

The middle term aim of this work is to create code with a guaranteed behaviour. It is therefore natural to start by checking the behaviour of a component, and then

to generate code-skeletons for the components.

6.1 Finding Abstractions

Defining abstractions can be burden without a tool support. For developping this kind of tool a first step is to characterise what is a good abstraction. It surely depends on the property to prove, but there are a couple of general ideas that support some automatising of the abstractions.

Using static analysis, the variables used in the property will signal which are the "variables of interest". The abstract domain for these variables is such that if there is a non-deterministic choice affecting the property, then the abstraction must be refined. There are tools like Bandera [14] that take this approach. Bandera defines a family of abstractions for a variable and lets the system find the least precise one that still holds the property. This work must be extended, though, to take into account futures. At least one needs to find the set of variables that may contain a future in any of their subfields. This leads us to the set of variables that must have a non-empty abstract domain as well. Moreover, this gives us the most abstract structure a variable can have for its type, i.e. a record with a field (or recursively subfield) for each of these variables with non-empty abstract domain.

6.2 Behaviour Model Generation

Building the behavioural model requires to abstract the JDC specification into a corresponding specification with only simple types. This is done by replacing each variable of user type by its abstraction. Then the pNets model will create:

(i) for each service, a storage for each of its local variables. A storage is a parameterized Labelled Transition Systems (pLTSs) that stores the variable state, and that exports actions *set* and *get* for accessing the variable. These storage are synchronised with all the pLTSs of the service methods and the service policy.

(ii) pLTSs for specific library elements of JDC, e.g. request queues, and proxies for futures. The latter requires dataflow analysis of the futures flow – in [3] we have defined a similar procedure.

(iii) a pLTS for each service policy. The service policy is a state machine so the transformation is straightforward. The *reactive* behaviour is transformed into two actions, one synchronised with the queue, and another that fires the affected service method. Similarly, each *active* behaviour is transformed into an action that fires the method directly.

(iv) a pLTS for each service method. This requires static analysis of the pseudo Java code of the abstract specification.

(v) synchronisation structures (pNets) for relating these pLTS. Each component is modelled by a pNet that synchronises the actions of the pLTSs – the model was previously shown in [2].

(vi) a tree of pNets modelling the architecture of the components. Each branch is the pNets model of a component, where its branches are the pNets of its subcomponents – the model was previously shown in [2].

6.3 Code Generation

From the JDC specification, it is possible to generate GCM/ProActive code-skeletons with the control code of the components. Java code is only generated for sequential components, so concurrent components must be provided with an architecture that decomposes the behaviour into sequential components. The ProActive middleware is adequate because it supports distributed components that communicate with first-order futures. We base our method on the following steps:

For each *Architecture* specification of a component, the compiler generates a composite component. The composite architecture is expressed with the GCM ADL (Architecture Description Language). The composite ADL defines the component type and its content based on the ADLs of other subcomponents, bindings and the IDLs of the interfaces.

Each *Service* denotes a sequential component, and therefore its natural implementation is a primitive component. An ADL is also created for defining the component type, as well as a reference to its Java implementation. A skeleton code is generated for the latter with the control flow. The code is a translation of the JDC's black-box specification based on:

- each service method in JDC is a public method of the component. We rely on the strong functional behaviour encapsulation of GCM for this matter, and that every possible method call and data-usage appears in the black-box specification.
- all data types are created, but these will need to be modified by the programmer to give implementation details.
- the service policy is implemented as a state machine within the ProActive's `runActivity()` method. This method dictates the initial activity of the component and we use it to orchestrate the access to the queue and to serve requests.

7 Conclusion

We aim at safe-by-construction components. Our approach is to define the architecture, the behaviour, and an abstraction of data within the specification language. The specification is formal enough in order to generate behavioural models that can be model-checked, and to generate code skeletons that include the control code of components.

More specifically, our contribution in this work is:

- A high level specification language for distributed software components, called JDC, that includes architectural, behavioural, and data parts. The behaviour of a component is given as a set of services; the details of a service are given in a Java-like language that makes easy to specify the control and data flow.

  The data part is an abstraction of the final application data classes. It must be designed by the developer as a compromise between verification and implementation concerns: precise enough to keep track of domains of variables affecting the control and data flow, but abstract enough to allow model-checking.
- Procedures for producing a hierachical behaviour model, in pNets format, on one side, and code skeletons, in GCM/ADL and Java, on the other side.

This work builds on the GCM, however, at the moment only a small subset of it is addressed. We plan to extend the language to cope with other interesting features, such as group communications and non-functional aspects (dynamic reconfiguration). Currently we have no tool support for the JDC language, except for a graphical version in the form of an Eclipse editor of the architecture part. Nevertheless, we plan to have a first prototype for the full language by the time of the workshop.

References

[1] Sensoria webpage. http://www.sensoria-ist.eu.

[2] T. Barros, R. Boulifa, A. Cansado, L. Henrio, and E. Madelaine. Behavioural models for distributed Fractal components. *Annals of Telecommunications*, accepted for publication, 2008. also Research Report INRIA RR-6491.

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

[4] BEA Systems, IBM, IONA, Oracle, SAP AG, Siebel Systems, and Sybase. Service component architecture. Whitepaper, November 2005.

[5] 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)*, LNCS 3054, may 2004.

[6] A. Cansado, D. Caromel, L. Henrio, E. Madelaine, M. Rivera, and E. Salageanu. *The Common Component Modeling Example: Comparing Software Component Models*, volume 5153 of *Lecture Notes in Computer Science*, chapter A Specification Language for Distributed Components implemented in GCM/ProActive. Springer, 2008. http://agrausch.informatik.uni-kl.de/CoCoME.

[7] A. Cansado, L. Henrio, and E. Madelaine. Towards real case component model-checking. In *5th Fractal Workshop*, Nantes, France, July 2006.

[8] 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*, 12(1):69–77, 2006.

[9] 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*, pages 123–134. ACM Press, 2004.

[10] R. Cleaveland and J. Riely. Testing-based abstractions for value-passing systems. In *Int. Conference on Concurrency Theory (CONCUR)*, volume 836 of *LNCS*, pages 417–432. Springer, 1994.

[11] A. Coglio and C. Green. A constructive approach to correctness, exemplified by a generator for certified Java Card appplets. In *Proc. IFIP Working Conference on Verified Software: Tools, Techniques, and Experiments*, October 2005.

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

[13] P. Cousot. Abstract interpretation based formal methods and future challenges, invited paper. In R. Wilhelm, editor, *Informatics — 10 Years Back, 10 Years Ahead*, volume 2000 of *LNCS*, pages 138–156. Springer-Verlag, 2001.

[14] M. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. Pasareanu, Robby, W. Visser, and H. Zheng. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23rd International Conference on Software Engineering*, 2001.

[15] F. Fernandes and J.-C. Royer. The STSLIB project: Towards a formal component model based on STS. In *Proceedings of the Fourth International Workshop on Formal Aspects of Component Software (FACS'07)*, Sophia Antipolis, France, September 2007. ENTCS.

[16] OMG. Corba components, version 3. Document formal/02-06-65, June 2002.

[17] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11), nov 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, September 2006. ENTCS.