A Component Oriented Notation for Behavioral Specification and Validation

Isabelle Ryl and Mireille Clerbout and Arnaud Bailly
L.I.F.L. CNRS UPRESA 8022
Bât M3, Cité Scientifique
59655 Villeneuve d'Ascq cedex
France
{ryl, clerbout, bailly}@lifl.fr

ABSTRACT

Component software development is definitely on a high trend in the sofware engineering field. However, integrating components which the producer does not have complete control over increases the risk of getting unexpected software behavior. So developing components for reuse by third-party integrators is a challenging task that one can make easier if the behavior of these software components is precisely specified.

In this paper, we introduce a specification language complementing the interface definition language IDL3 proposed by OMG to describe CORBA Component Model compliant components. This specification language is based on communication history : the sequence of observable events - method calls, return of method calls, events, exceptions – that occurred since the system has been started. It allows us to characterize the functional behavior of components by way of invariants : an interface invariant specifies a contract between a component that provides it and each of its clients, whereas a component invariant constraints the whole communication between one component and all its clients and servers. We propose a procedure based on this notation for generating specification-based test cases adapted to unit testing and discuss how to use this notation for validation purposes.

1. INTRODUCTION

The development of component based applications is clearly a necessity for software industry in a world of large scale distributed applications. Furthermore, the growth of a market of reusable components, the famous *Components off-the-shelf* – COTS – seems to be the only way to reduce production costs in a domain where constantly evolving technology checks productivity. Nevertheless, this approach makes sense if components are high-quality and really "composable". This can be achieved by strict development methods and interoperability norms. This last point is addressed by platforms like CORBA, EJB or DCOM.

The well-known advantages of a "component oriented" approach are the following :

- *reusability.* Components are "bricks" that are available for designers,
- *maintainability* . Functionalities can be added or modified just by insulating responsible components,
- *interoperabilty.* Components implementing a given specification can be searched amongst existing components offered by various providers on the COTS' market.

To reach these goals, the specification of components has to be precise enough to allow searching, testing and composing components which are potentially designed, implemented, integrated, and used by different people. The mere existence of a market itself depends on the confidence providers and consumers may put in the functional correctness of a product.

In "Component Software" [22], Clemens Szypersky proposes the following definition of components: "A Software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties." This definition emphasizes the use of well specified interfaces to describe components because the interfaces have to be the only link between a component and its outer-world i.e. its clients and environment.

One way to correctly specify interfaces is to use specific interface description languages, like the OMG-IDL for CORBA objects and the new OMG-IDL3 for CORBA Component Model compliant components. IDL typically defines contracts between components using method signatures. Unfortunately IDL contracts are mostly syntactic and do not capture the semantic aspects of the contract. This leads to under-specified components and breaks the initial interoperability goal. As an example, we can mention the work of Ousmane Sy [21] who showed that five different implementations of the COS event service by CORBA framework

providers were neither inter-operable nor substitutable because of diverging interpretations of the OMG-IDL specification.

During the last few years, the interest in formal software development has grown steadily. Approaches based on formal methods provide the advantages of a precise and non-ambiguous specification language with a formal semantic. And, even if formal proofs are out of their scope, formal verification is possible and generally supported by tools.

Sofware quality should benefit from formal method based component development a various stages:

1. a formal specification language has to be rich enough to express the full semantics of components,
2. powerful tools have to be able to check the consistency of specifications, using for example type-checking. Higher level properties must be verified: it may be possible to control that a component respect the contracts of its required environment, to control that the implementation of a component conforms to its specification and that connected components are compatible.

To fulfill this requirements, we propose to increase the specification power of OMG-IDL3 from the OMG CORBA Component Model (CCM) with formal specifications. This increased specification will follow the component through its life-cycle and will serve as *lingua franca* for all the actors – designers or users – to describe the behavior of the component. We only focus on describing functional aspects and do not address non-functional requirements like quality of service or overall reliability.

To us, a component is a black box that offers services, requires a specified environment, and can emit or receive asynchronous events. Interfaces are seen as contracts between two components or, in a more practical point of view as communication channels (like the event channels) between components. A system is then a set of components which are connected to each other by interfaces and event channels, and we focus on communications between components. Our specification language allows us to express two kinds of property. The first one is the notion of protocol, that is to say the partial order in which the input and output communications have to occurred. The second one is the data level, i.e. the possibility to specify properties on the value of an input or output parameter.

To meet these requirements, we propose a specification language based on communication traces.

The basic events of the system we consider are the messages components exchange, which are supposed to be instantaneous and asynchronous:

- method calls between interfaces of components,
- return of method calls,
- exceptions between components,
- event sending,
- event receiving.

Note that internal method calls are not considered here.

The trace of a running component, system or sub-system is the sequence of all the events involving it that occurred since its creation. The purpose of the specification is to capture the set of all possible communication traces of a given system. Note that a lot of works deals with the trace notion in different contexts (for example CSP [10], trace theory [8], ...).

After a short introduction to the CCM abstract model and IDL3, Section 2 describes our specification language. Section 3, gives some ideas of validation activities and proposes a way to test components. After a short review of related works, we conclude with some possible further work.

2. HOW TO SPECIFY COMPONENTS?

As already said, we build upon the CORBA Component model. We insisted on the fact that the specification could increase the reliability of the component at several stages in its life. Therefore, the specification ought to be present every time it might be needed and this explains our choice of IDL3 as our ground language. It offers two advantages:

1. IDL3 already exists, is used by several tools, and is embedded in the component package,
2. the IDL3 language contains all the necessary syntactic definitions of components and interfaces that can be exploited for formal specification purposes.

In order to respect the IDL3 syntax, we just insert some formal properties of components as IDL3 comments. That way, the specification is embedded in the packaged component and do not interfere with existing tools. To enlighten our methodology, we will develop in the remainder of this article a small example: an electronic ballot system.

Example informal specification. *Each voter uses an electronic individual polling device that communicates with a central office which registers the votes. To simplify the writing, we just consider a referendum and some rules:*

- *voters are allowed to vote at most once,*
- *voters are allowed to read the results when the ballot is over,*
- *the central accepts votes until the closure, then rejects them,*
- *the central refuses to provide partial results before the closure,*
- *the central must announce the correct results !!!*

The first subsection presents the general CCM model, the second one briefly introduces the IDL3 syntax. The last one presents our formalism on the example.

2.1 Model

To formally specify software components, we first have to agree on the definition of a component. We try in this subsection to present the abstract model we use that is part of the CCM model. We are just a little bit more restrictive

than the CCM ; CCM components may support interfaces and components may communicate with other components outside the scope of interfaces: this is not allowed in our model where all the components must declare ports (interfaces) and communicate through these ports only.

Interfaces are used by developers as implementation contracts for components and by clients to interact with components at runtime. Interface definition contains attributes and method signatures and may be defined using multiple inheritance. An interface groups a – hopefully coherent – set of methods providing some services and, according to the connection mechanism of the CCM model, it becomes a communication channel for data exchange between components.

A **component** definition groups attributes and ports. Attributes are properties of components set at deployment time for configuration purpose. Attributes misuse may raise exceptions. Ports come in four flavors:

- *facets* are interfaces provided by the component and synchronously used by the clients,
- *receptacles* are interfaces used by the component in a synchronous mode,
- *event sinks* are interfaces provided by the component and asynchronously used by the clients,
- *event sources* are interfaces used by the component in an asynchronous mode.

Ports are used at deployment time and runtime to connect components together. According to this model, a component may be seen as an element of a system that offers a collection of services under the assumption that another collection of services is available. This approach makes component exchange easier. Note that components may be defined using single inheritance (in a Java style). Figure 1 shows the diagram (drawn from [15]) used to represent a component and its ports.

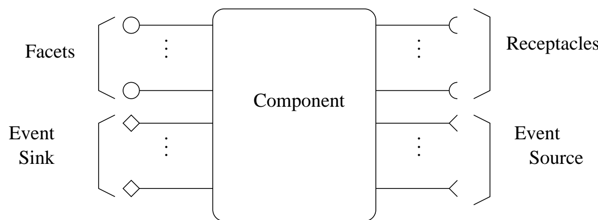

Figure 1: Component.

A **system** is a set of components. The current set of connections between components represents the system's configuration. Figure 2 presents our proposed model for the electronic vote example. Two types[1] of components are used, one for the votes collecting central office, **Center**, and one for the individual polling devices, **Electronic_box**. An **Electronic_box** component is used by each voter, it offers an interface *Electronic_vote* that allows the voter to send his vote or get the results. **Electronic_box** components are distributed and connected to the *Vote_Center* interface of a **Center** component that centralizes information. **Center**'s main role is to count votes, it also offers an interface *Vote_Admin* used to close the ballot. When the ballot is closed, the **Center** sends an event to inform connected components of the closure and transmits them results. The event sinks of **Electronic_box** components are connected to the event source of **Center**: the events sent via this source will be received by all connected components.

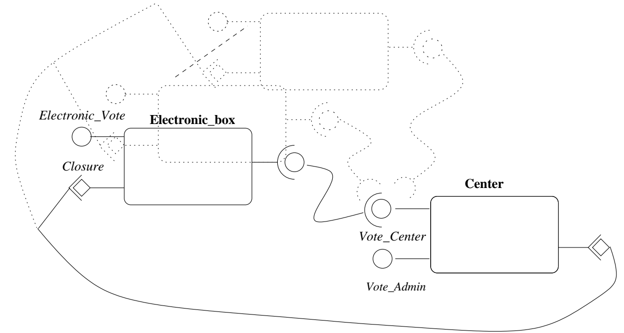

Figure 2: The example model.

2.2 IDL3 short presentation

The OMG IDL3 language extends IDL2 with some syntactic constructs to take into account the CORBA Component abstract model. This section shall introduce IDL3 concepts relevant to our example and is by no mean a full description of the CORBA Component Model. We refer the reader to [15] for more details about the CCM.

A part of the syntax is inherited from IDL2. Declarations are gathered in modules that delimit a domain name. Interfaces declare attributes and methods. Interfaces may be defined using multiple inheritance. Declarations are in a C/Java style, methods may have `in`, `out`, and `inout` parameters of simple types (`Short`, `Long`, `Float`, `Char`, `String`, `Boolean`, ...), complex types (`Enum`, `Struct`, `Array`, ...) or object types. Methods may also throw exceptions which may be defined by the user.

Figure 3 shows the IDL3 definitions of our example's interfaces. The module `Vote` starts by defining four exceptions (an exception may contain fields) followed by interface definitions. `Electronic_Vote` is the interface used by the voters, it offers two methods, one to vote and one to read the results. The first one has an in-parameter which represents the value of the vote (`true` for "yes" and `false` for "no") and it may throw two exceptions: `too_late` if somebody tries to vote after the closure and `already_voted` if somebody tries to vote twice. The method `read_results` has two out-parameters respectively returning the number of "yes" and "no" votes of the ballot. This method may also throw an exception when used before the closure (the results are not available). The interfaces `Vote_Center` and `Vote_Admin` are facets of the `Center`, they are respectively used by the `Electronic_Box` components and by an administrator. The `vote` method is used to transmit the elector's vote to the

[1] For lack of space, we do not address the problem of identification of voters which can be solved using a classical "login/password" protocol.

center, it returns a boolean indicating if the vote has been taken into account or not (in case of closure for example). The method `close` allows an administrator to close the ballot and throws an exception if already closed. Comments are between `/**` and `*/` or `//` and end of line.

```
module Vote {
   exception too_late{};
   exception already_voted{};
   exception already_closed {};
   exception not_closed {};

   interface Electronic_Vote {
      void vote (in boolean choice)
               raises (too_late, already_voted);
      void read_results (out long yes,
                         out long no)
               raises (not_closed);
   /**
      invariant
         (h;_<-this.vote(_) |- H)
               => (!_<-this.vote(_) in h)
         &&
         (h;_<-this.vote<already_voted> |- H)
               => (_<-this.vote(_) in h)
         &&
         ((_<-this.vote<too_late> |
         _<-this.read_results (_,_));h in H) =>
               h/<- |- (_<-this.vote<too_late> |
               _<-this.read_results (_,_)))*
      */
   };

   interface Vote_Center {
      boolean vote (in boolean choice);
   };

   interface Vote_Admin {
      void close() raises (already_closed);
      /**
      invariant
         (h;_<-this.close() |- H) =>
               (!_<-this.close() in h)
      */
   };
   ...// insert here component declarations
};
```

Figure 3: Interface Specifications.

Components are introduced by the keyword `component`. As already said, components may contain attributes and ports definitions. Several keywords allow us to distinguish different kinds of ports: `provides` and `uses` for facets and receptacles, `consumes` for event sinks and `publishes` or `emits` respectively for *1-to-n* or *1-to-1* event sources.

Figures 4 and 5 complete our example's definition. Component `Center` has two facets and one *1-to-n* event source while component `Electronic_box` has one facet, one receptacle and one event sink. Each producing or consuming event port specifies a type of event: events are simply valuetypes (that is to say objects by value) that inherit from the `Components::EventBase` interface. In our example, a closure event has two attributes encoding results of the ballot: `yes_number` for count of "yes" and `no_number` for count of "no". Thus, an event has a meaning of its own (here indicating ballot's closure) but may also carry values (here results of the vote).

One can notice that all ports of a component are named and that a component may offer several ports of the same type. The (de-)connection operations are relevant facts in the component life, they consist in the assignment of real references to receptacles and sources of the component. Connection and deconnection operations are implicit methods of any component.

2.3 Specification language presentation

We have previously expressed our goals, and said that we use IDL3 as our components' definition language. It is now time to describe the formal specification language inserted as comments. As said earlier, we only focus (at least at this stage of our work) on the functional aspect of components, aiming to answer to these questions :

- How to describe a component's behavior ?
- How to ascertain that two components are interchangeable *i.e* they offer the same services?
- How to be sure that several components can be connected?

A system evolves from its creation to its destruction by way of components interactions. Thus, the "state" of a system (or component) may be seen as the result of its past interactions with its environment. So, the specification language we use is based on the concept of communication "history". This idea comes from an object oriented formal notation developed at Oslo University: OUN (Oslo University Notation)[17].

The communication history of a system is a sequence of observable events (a trace) that records all communications between components that occurred since the system has been started. The notion of "observable" event depends on the viewpoint we consider:

- observable events of a system are all the communications between components (method calls, returns of method calls, events, exceptions),
- observable events of a component are the events of the system that can be "seen" from the viewpoint of the component, that is to say all the communications whose sender or receiver is the component,
- observable events of an interface of a component are the communications involving the component through this interface.

Events of the system are supposed to be instantaneous, they may fall into four categories:

```
component Center {
   provides Vote_Center v;
   provides Vote_Admin a;
   publishes Closure c;
   /**
   invariant
         (h;_<-a.close() |- H) => (!_<-a.close() in h)
      && (h;_<-a.close<already_closed> |- H) => (_<-a.close() in h)
      && (h;_<-v.vote(_:true) |- H) => (!this->c[x,y] in h)
      && (h;_<-v.vote(_:false) |- H) => (this->c[x,y] in h)
      && (h;_<-a.close() |- H) => (h1;this->c[x,y] |- h && x=result(h1,true) && y=result(h1,false))

   functions
      result :  Trace, boolean -> int;
      result(empty,_) -> 0,
      result(h;_<-v.vote(x),x) -> result(h,x) +1,
      result(h;_, x) -> result(h).
   */
};
```

Figure 4: Component `Center`.

1. $c \rightarrow (c' : i).m(\bar{x})$ denotes a call of method m with the tuple of parameter values $\bar{x}$ initiated by component c, m is a method of facet i of component c'. Events belonging to this category are termed initiation events,
2. $c \leftarrow (c' : i).m(\bar{y} : z)$ denotes "normal" return of a previous event, the parameter values $\bar{y}$ may be different from $\bar{x}$ since out and inout parameter modes are allowed. The optional $: z$ value denotes possible return value of the method. Events belonging to this category are termed termination events,
3. $c \leftarrow (c' : i).m < e >$ denotes termination of a method call by an exception e. Those events are termed exception events,
4. $c \rightarrow \{(c_1 : s_1), \ldots, (c_n : s_n)\}[\bar{x}]$ denotes an asynchronous message, an event sent by component c to sinks $s_1, \ldots, s_n$ of components $c_1, \ldots, c_n$ respectively with tuple of values $\bar{x}$. Those events are termed asynchronous events.

Note that we do not characterize attributes and (de-)connection events: we consider attributes as pairs of set/get methods and (de-)connection events as method calls.

The set of possible events of the system is called the **alphabet of the system** (it depends on interfaces and components comprising the system). We can define in the same way alphabets for various elements of the system depending on what they can observe.

The **alphabet of a component** is the subset of events of the system whose receiver or sender is this component.

The **alphabet of a component seen through an interface** is the subset of the component alphabet containing events that are initiation, termination or exception of method calls defined in this interface.

A **trace** of a component (resp. a system) is a sequence of events of this component's (resp. system) alphabet in which a termination or exception event responds to a past initiation event. Notice that asynchronous events have no corresponding terminations. One may think of a trace as a sequence of events of a running system registered in the order they appeared since the start of the system until an arbitrary observation instant. Clearly, the system may run after the observation – a trace does not represent complete execution – and observation may occur at any time – a trace's prefix is also a trace.

A system element's specification describes its behavior in terms of possible communication traces. A component's specification is expressed as an invariant on traces (in first order logic). This invariant characterizes a subset of all possible traces over the component's alphabet hence capturing this component's semantic.

An **invariant** may be added to each interface or component. Informally, it is a first order formula with variables, constants, functions, predicates and constraints whose models are the valid traces of the component.

Our purpose is to convince the reader of expressiveness of such a notation and not to detail the syntactic sugar it offers, so we give an idea of the notation on the example.

Interface invariants in our example are shown in Figure 3. In our model, interface invariants are seen as *1-to-1* contracts, so they specify communications between a component implementing the interface, represented by the keyword `this`, and one of its clients, represented by the symbol _ which denotes any value in the corresponding domain (here the set of components). This does not inhibit one-to-many communication schemes: a component may receive method calls from different clients through one of its interfaces but

```
component Electronic_Box{
   provides Electronic_Vote e;
   uses Vote_center v;
   consumes Closure c;
   /**
   invariant
        (h;_<-e.vote(x) |- H) => (this->v.vote(x);this<-v.vote(x:true) -| h )
      && (h;_<-e.vote<too_late> |- H) => (_->c[_,_] in h)
      && (h;_<-e.read_results(x,y) |- H ) => (_->c[x,y] in h)
      && (h;_<-e.read_results<not_closed> |- H) => (!_->c[_,_] in h))
   */
};
```

Figure 5: Component `Electronic_Box`.

the invariant of the interface specifies the communication pattern with each client. Clients are independent from each others thus the contract declared in each interface must ensure that clients can use some services whatever other clients may do. For example, if a file must be opened by a client before being read, a client following this rule must not be affected by another client trying to read this file without opening it.

Let us first detail the `Electronic_Vote` interface. The symbol `H` denotes the history, that is to say a solution to the formula. This invariant is a conjunction (denoted by `&&`) of three implications. The first one says that a vote ends correctly if it is the first one: if a sequence (denoted by `;`) of a trace `h` and the termination of a `vote` is a prefix (denoted by `|-`) of the history then `h` does not contain any termination of `vote` (`!` denotes the negation and `in` denotes "is a factor of"). The second part of the invariant describes the converse situation: a vote ends by exception `already_voted` if it is not the first one. The third part addresses another problem: votes are accepted before the closure and results are available after the closure. The closure may occur at any time, the client of the interface cannot detect it except if he receives a `too_late` exception or the results. This part of the invariant says that any part `h` of the history following an exception `too_late` in response to a vote or (denoted by `|`) a termination of `read_results`, only contains responses to method calls that are exceptions `too_late` or terminations of `read_results` (`/<-` denotes the projection[2] onto termination and exception events only).

Interfaces `Vote_Center` and `Vote_admin` have simpler specifications. There is no invariant in the first one since there is only one method whose use does not depend on anything observable in this interface. We could have said that as soon as the return value has been `false`, it remains `false` until the end (the vote is closed) but the user is free to specify properties or not if they seem not to be useful. The invariant of `Vote_Admin` just says that a closure operation ends normally if it is the first one: it is not possible to close the ballot twice. The invariant does not specify at what point the exception `already_closed` may occur because it is not decidable in the interface: a client cannot predict if another client has closed the vote before him.

[2]The projection of a trace t onto a set of events E can be seen as the operation that deletes in t all the events that do not belong to E.

In this example, we do not speak about initiation events: the component implementing the interface is not responsible for input events, it may only ensure its own outputs. Inputs events are often used to specify several situations of the kind: "if a client sends me this event before this one then this will happen ...".

Component invariants follows same syntax than interface invariants, but a component invariant specifies the whole communication pattern between this component and all other components, clients or servers: at this stage it is possible to specify the interleaving of its communications with several other components. The trace set of a component is described by its own invariant and the invariants of the interfaces it provides.

Definition. (Trace set) *Let us denote by Σ_c the alphabet of a component c and $f_1, \ldots, f_n$ its facets. The invariants of the component and the interfaces are respectively denoted by $\varphi, \varphi_1, \ldots, \varphi_n$. Then, the trace set of the component c is defined by:*

$$\{t \in \Sigma_c^* \mid \varphi(t) \wedge \forall i \in \{1, \ldots, n\}, \forall c' \bullet \varphi_i(t/f_i/c')\}$$

where c' denotes another component and $t/f_i/c'$ the projection of t over the events that are communications through the facet f_i with c'.

In other words, the trace set of c is the set of words defined on Σ_c that satisfy the invariant of c and the invariant of each facet of c with respect to a projection over communications with another component through this facet[2]. Note that there is a subtle difference between the alphabet's definition and the notation used: as soon as two components are connected, the receptacle contains the reference of the interface of the other component, so it may be used directly. Thus, we use the notation `c->r.m(x̄)` instead of `c->(c':i).m(x̄)` when the receptacle `r` of `c` is connected to the facet `i` of `c'`. Similarly, we do not denote the current component by `this` but by the name of the port involved in the communication, this allows us to easily distinguish communications on different ports.

Figure 4 gives the specification of the Center component, a conjunction of five implications. The first and the second ones say that the ballot may be closed at most once, otherwise an already_closed exception is thrown. This invariant differs from Vote_Admin interfaces' invariant because of the possible instantiations of mute symbol _: in the component invariant, it may represent any other component each time it appears. Thus we can say that a client calling the close method receives an exception if the termination of close has already occurred, whichever client has received this termination. The two following implications say that votes are accepted while the closure has not been announced and that they are rejected (return value false) as soon as the closure is announced. The last implication is a little bit different since it uses a function. The "functions part" of the specification is an auxiliary part that allows the user to define its own functions for specification purposes only. The definition of a function must start by the name of the function, and the parameter and return value types. The function itself is defined in a Prolog style by several clauses: the clause that will be used is chosen by unification on the heads of clauses in the order they are declared. Note that functions do not have side effects on traces or values passed to them. The result function calculates the result of the vote on a trace of the component: it takes two parameters, a trace and a boolean and it returns number of votes that appear in the trace with the boolean as parameter value. If the trace is empty, the result is 0 whichever boolean value is given as input. If the trace ends by a vote termination event whose parameter is the boolean which is currently counted, the result is the function applied to the beginning of the trace plus one. If the trace ends by any other event, this event does not affect the result. Let us return to the last part of the invariant. If a close event succeeds, then an asynchronous event has been sent with values x and y that are the correct results of the vote. This example shows how powerful is the specification language: it is possible to describe when calls or exception occur but also to precisely calculate parameters' values.

The form of the specification of the Electronic_Box component is very similar to the previous one, we only detail the first part of the conjunction in which three components are involved. It says that voters' choices are correctly transmitted to the center: each termination event of vote is immediately preceded (-| is read as "is a suffix of") in the trace by center's method vote initiation event (*i.e.* vote) with the same value of the vote x and matching termination with return value true. Thus, the vote has been transmitted and, since the three events are consecutive, we can deduce that there is one transmission for each vote.

To conclude on this example, the language allows us to express properties like protocols between several components as well as precise descriptions of parameter values and case when exceptions are thrown. The example of the function result shows that it is possible to calculate, using a function, an abstract state of a component from the trace. Our model supports synchronous (method calls) as well as asynchronous (events) communication. Components may be implemented using multi-threading, re-entering code and so on. Anything concerning the implementation is out of our scope so for example we cannot express the fact that a component must be multi-threaded, but our model supports it.

3. V&V ACTIVITIES

As said in the introduction, our project aims at providing tools to exploit the specification at different stages of the component's life. A formal specification provides a strong reasoning basis to deal with the system's properties. The specification is all the more useful as automated tools are provided. We propose to use the specification for different purposes: to validate the specification, to ensure that the component is conforming to the specification by way of testing methods, to check the composability and the substitutability of components. We expose in this section the different ideas, emphasizing the test phase which is the most advanced part of the work.

3.1 Validation

Specification Validation. The first step is to ensure that the specification satisfies the user's requirements. A specification written by a user is not systematically valid and may contains two kinds of errors:

- errors using the language concepts that may lead to inconsistency — *e.g.* an empty trace set or a trace set not closed under prefix,
- design errors that may lead to under specification.

In order to detect such errors, validation tools must be provided to check the specification. Clearly, the general problem of the validation of specifications is undecidable so any tool we provide will not be completely automatic.

To address this problem, we benefit from the work initiated in Oslo for the OUN notation [17]. Even if the two notations are not exactly the same (there are objects in OUN and components in our notation and it is syntactically more restrictive) the basic concepts are very close so we can use the same approach. The solution adopted in OUN is to use the tools offered by the PVS toolkit [18, 6]: PVS provides (among other things) a model-checker and a powerful theorem prover. The particularity of the prover is to allow the user to include proof strategies adapted to its own problems: this increases the automation of the proofs. The idea is to define the semantics of OUN in PVS in order to directly use the PVS tools, the work of [13] may be adapted to our notation.

Testing. As soon as the specification satisfies user's requirements, the problem is to obtain an implementation of the component that conforms to this specification. One approach is to generate code from specification, this produces a safe code but we rejected it for several reasons. First, the goal of the "component approach" is to free the developer from technical stuff to make him concentrate on business logic: specialists are recognized to write efficient code adapted to their domains. Second, our approach considers components as interchangeable "black boxes" and peculiarities of the code are out of our scope. As most of the software production lines do, we propose to use testing to check a component's correctness with respect to its specification. The test process is detailed in the next subsection.

Another problem is **composition**: how to be sure that two components are compatible ? Once components are shown to conform to their specifications, the compatibility of two components depends on the compatibility of their specifications. The effective connections of components are dynamical but port types may be used to statically check the correctness of the possible connections during the validation phase of the component (if there is one) or later during the assembly phase. It suffices to check that each component respects the contract of interfaces it uses and that each used interface provides the expected services[3]. For that, we have to check the compatibility of trace sets. A component c will behave correctly when assembled if (1) its traces relative to the viewpoint we consider (here by the way of projection) satisfy the invariant of the interfaces it uses and (2) if the used interfaces do not provide unexpected outputs (non-deterministic components may have several outputs for the same inputs). In other words:

Definition. (Connectable components) *Let c be a component and $\mathcal{T}_c$ be its trace set. The component c is "connectable" if and only if for each type I of receptacles or object parameters of c, for all component c' providing a facet of type I, and for all trace t of $\mathcal{T}_c/I/c'$:*

$$\varphi(t), \quad (1)$$
$$\forall h \in \mathcal{T}_{c'}/I/c \bullet (h/ \to = t/ \to) \Rightarrow (h \in \mathcal{T}_c/I/c'). \quad (2)$$

This definition is supposed to ensure compatibility of components. This is the case when receptacles and parameters used by a component are exactly of the declared type. The following question concerns sub-typing: what happens if we connect a receptacle of type I to a facet whose type is a sub-type of I? We expect all static verifications to remain valid whatever dynamic connections are made. For that, we use behavioral sub-typing: a subtype will behave like any of its super-types in the same context. Thus, the definition of connectable components we gave is valid even if sub-typing is used and we have a strong inheritance relation. "Behave like super-type" means for us that if we give the inputs of the super-type to the subtype, the subtype produces outputs that could have been produced by the super-type:

Definition.[4] **(Behavioral Interface Inheritance)** *Let I be an interface that inherits from $J_1, \ldots, J_n$. Let the formulas $\varphi_1, \ldots, \varphi_n$ be the invariants of $J_1, \ldots, J_n$ respectively. The inheritance relation of I is correct if and only if for all component c providing I and for all component c':*

$$\forall t \in \mathcal{T}_c/I/c', \forall i \in \{1, \ldots, n\}[(t/ \to = t/J_i/ \to) \Rightarrow \varphi_i(t)].$$

The correctness of the inheritance relations of interfaces should be proved during the validation of the specification. Note that component inheritance does not affect our verifications so it is not constrained.

The specification may also be useful at other stages in a system's life. We can for example evoke the maintenance: when can we say that two components are interchangeable ? Since our model pays a large attention to interfaces, we can assert that two components having the same ports are strictly **equivalent** (we still speak about functionalities). This relation may be too strong from a practical point of view: it is interesting to replace a component by another one, different but providing at least the same services in the same context, we say that they may be substituted:

Definition. (Substitutability) *A component c' may be substituted for a component c if:*

- *the services provided by c may be provided by c'* i.e. *for each facet of c, either c' provides a facet of the same type or it provides a facet of a behavioral subtype,*
- *the services required by c' are available* i.e. *receptacles of c' are of the same types or behavioral super-types as receptacles of c,*
- *c and c' have the same sources and sink.*

This definition works because of the behavioral inheritance relation we defined earlier on interfaces. In some sense it defines a kind of behavioral sub-typing for components.

3.2 Component testing

We have already explained the reasons which lead us to chose a testing method to verify correctness of a component towards its specification. In this subsection, we give some more details about the testing process. Defining a testing procedure requires to:

- select a representative set of test cases since exhaustive testing is not tractable,
- have an oracle, that is to say something (another program, a human, a specification, ...) able to decide if the program gives the right answer when executed with a test case,
- be able to execute the tests, that is to say run the tested program with the chosen test cases (inputs) and check (using the oracle) the program's outputs correcteness.

Our proposition takes place in functional, unitary, and specification-based black-box testing. The use of formal specifications for testing has several well-known advantages, and because of its form, the notation we propose has the following ones:

- we use it to generate test cases which are traces, thus we test a complete behavior involving several other components, several methods, client and server aspects of the component, and not a simple stimulus/response protocol,
- the specification is the oracle because traces contain inputs as well as outputs. Note that the specification language form is very different from programming languages, thus avoiding redundancy errors.

Test-case generation. As a first step, we have to generate a representative subset of a component trace set. The form of the specification introduces two kinds of problems: first

[3] As a matter of fact the same checks are necessary for each object parameter, the process is identical.

[4] Using this definition, specifications are not inherited. This choice gives some freedom in the sub-typing relations but may lead to excessive writing work. We could add a stronger inheritance relation using projection that would add more constraints on sub-typing but offers specification inheritance.

we have to generate sequences of events and secondly we have to find parameter values that satisfy the constraints.

The invariant is a formula: terms and logical connectives. First, we consider the formula from the viewpoint of propositional calculus where trace predicates are the atoms of the formula. By finding all combinations of predicates that satisfy this formula we are defining all possible behaviors of this component, which is close to the classic disjunctive normal form partitioning techniques from the test literature [7]. Each solution gives us formulas to obtain sets of test cases. The problem is then to generate these test cases. The terms generate languages that contain variables constrained by predicates. For example, the invariant of `Electronic_Box` is a conjunction of implications. The resolution gives us the disjunctive normal form we could have obtained by replacing each $A \Rightarrow B$ by $(A \wedge B) \vee (\neg A \wedge B) \vee (\neg A \wedge \neg B)$ and distributing. So, for example one term of the disjunction is given by Figure 6.

```
   (h;_<-e.vote(x) |- H)
∧ (this->v.vote(x);this<-v.vote(x:true) -| h )
∧ ¬ (h;_<-e.vote<too_late> |- H)
∧ ¬ (_->c[_,_] in h)
∧ ¬ (h;_<-e.read_results(x,y) |- H )
∧ ¬ (_->c[x,y] in h)
∧ ¬ (h;_<-e.read_results<not_closed> |- H)
∧ (!_->c[_,_] in h))
```

Figure 6: A sample conjunction of Invariant terms.

The solution we adopted to obtain the traces is to use Prolog as introduced in [14]: operators on trace languages, predicates, functions defined by users (which are already in a Prolog-like form), ... may be defined as Prolog clauses. The goal of the program has one free variable which is `H` thus, it enumerates the traces. To obtain a tractable test case set, we have to define when to stop. We use the regularity hypothesis introduced in [3] which formalizes the following idea: "if we test all the test cases whose complexity is lower than the complexity of the formula, then we can consider that the formula is valid". The problem is to define the complexity to use. For the moment, we consider the number of operators allowing to generate the traces which is not fully satisfying. When the sequences are built, the last operation consists in instantiating parameter variables. Constraints on parameters are given by predicates and we use uniformity hypothesis introduced in [3] to select one representative value for each domain.

Test-case execution. We obtain a set of test cases which are traces, the problem is now to execute the traces. The abstraction level of our model gives us traces that are not directly executable. During the process development of components, the IDL3 description is projected in IDL2 and then some rules define projections onto different programming languages. So we have to apply the same projections rules to our traces to obtain the real traces that may be observed during the component execution.

The development of a test platform aimed at EJB is in progress. The basic idea is to use a test container. A component under test is placed in the test container which monitors the test executions: other components are represented by stubs that transmit messages to the container.

4. RELATED WORKS

There is now widespread acceptance over the necessity to specify software system and one acknowledges that having a precise and unambiguous formal specification available is a prerequisite in order to automate black-box testing.

Today there exist some proposals for specification languages designed for Interfaces Definition Languages : for example Larch/Corba [20] which is rather data-oriented due to its roots in Abstract Data Type or Borneo [19] where constraints do not consider data. Note that these two proposals only take into account server aspects of components.

In [5, 4] the authors extend general IDLs and then CORBA-IDL with protocol information concerning supported and required services using Milner's polyadic π-calculus which seems to be a more low level syntax language than ours and [2] uses a formalism based on Petri Net to specify CORBA component. The specification conformance with testing is not addressed in these works.

The generalized use of UML notation and its various behavioral formalisms (state-charts, collaboration and sequence diagrams) brings out several approaches (and tools) for test-case generation in object-oriented software. These works are based on techniques closed to finite state machines or finite labeled transition systems [9, 16, 12, 11].

Another class of approaches for test generation uses partition testing techniques [3, 7, 1]. Even if these techniques generally lean on model-based or algebraic specifications our project will benefit from these works.

5. CONCLUSIONS

We have presented in this paper a component oriented formal notation and some of the general definitions that can be exploited to validate specifications and to prove component properties. This work takes sense if tools are provided to specify and validate implemented components. The work is in progress, especially the test platform for EJB. Since Einar Broch Johnsen recently finished the definition of OUN semantics in PVS [13], we are now ready to exploit it to deal with validation aspects. Some other aspects could impact our work in the future. The first one is to address the general composition problem: how to build a component from components? In our context, the problem is to find a trace set for the assembly from the component trace sets, it relates to the problem of formal language reconstruction. The second one is to describe a component which an application needs and find it. This requires a specification of the application and to be able to extract from it a component specification (which is not the most difficult part). Clearly, the major problem would be to search a component and the automation of the search seems not to be tractable for the moment. Anyway, if a candidate component is found by a user, it is possible to compare its specification – if it has one – to the specification deduced from the application's or to test it – if it has no specification, in order to be sure that it is convenient for the application.

6. ACKNOWLEDGMENTS

We are grateful for feedback from discussions with the GOAL team, in particular Raphaël Marvie and Philippe Merle. Jean-Marc Geib has provided valuable detailed advice concerning this manuscript.

7. REFERENCES

[1] B. K. Aichernig. *Systematic Black-Box Testing of Computer-Based Systems through Formal Abstraction Techniques*. PhD thesis, Technischen Universität Graz, Germany, 2001.

[2] R. Bastide, O. Sy, and P. Palanque. Formal specification and prototyping of CORBA systems. In *Proc. ECOOP'99, Lisbon Portugal*, vol. 1628 of *LNCS*, pp. 474–494, 1999.

[3] G. Bernot, M.-C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *IEEE Software Engineering Journal*, 6(6):387–405, 1991.

[4] C. Canal, L. Fuentes, J. Troya, and A. Vallecillo. Extending CORBA interfaces with pi-calculus for protocol compatibility. In *Proc. TOOLS Europe'2000), Mont Saint-Michel, France*, pp. 208–225. IEEE Computer Society Press, 2000.

[5] C. Canal, L. Fuentes, and A. Vallecillo. Extending IDLs with pi-calculus for protocol compatibility. In *Proc. ECOOP'99 Workshop Reader, ECOOP'99 Workshops, Panels, and Posters*, vol. 1743 of *LNCS*, pp. 5–6, 1999.

[6] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. In *Proc. Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, Florida, 1995.

[7] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Proc. FME'93: Industrial-Strength Formal Methods*, vol. 670 of *LNCS*, pp. 268–284, 1993.

[8] V. Diekert and G. Rozenberg, editors. *The Book of Traces*. World Scientific, Singapore, 1995.

[9] J. Hartmann, C. Imoberdorf, and M. Meisinger. UML-based integration testing. In *Proc. ISSTA 2000*, pp. 60–70, Portland, Oregon, 2000.

[10] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[11] T. Jéron, J.-M. Jézéquel, and A. Le Guennec. Validation and test generation for object-oriented distributed software. In *Proc. PDSE'98, Kyoto, Japan*, 1998.

[12] J.-M. Jézéquel, A. L. Guennec, and F. Pennaneac'h. Validating distributed software modeled with the Unified Modeling Language. In *Proc. UML'98 - Beyond the Notation, Mulhouse, France.*, vol. 1618 of *LNCS*, pp. 365–377, 1998.

[13] E. B. Johnsen and O. Owe. A PVS proof environment for OUN. Research Report 295, Department of Informatics, University of Oslo, june 2001.

[14] B. Marre. *Une méthode et un outil d'assistance à la sélection de jeux de tests à partir de spécifications algébriques*. PhD thesis, Université de Paris-Sud – Orsay, 1991.

[15] R. Marvie and P. Merle. Corba Component Model: Discussion and use with OpenCCM. *Informatica*, submitted.

[16] A. J. Offutt and A. Abdurazik. Generating tests from UML specifications. In *Proc. UML99, Fort Collins, CO*, pp. 416–429. IEEE Computer Society Press, 1999.

[17] O. Owe and I. Ryl. A notation for combining formal reasoning, object orientation and openness. RR 278, Department of Informatics, University of Oslo, 1999.

[18] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, 1995.

[19] S. Sankar. Introducing formal method to software engineers through OMG's CORBA environment and interface definition language. In *Proc. AMAST'96 Munich, Germany*, vol. 1101 of *LNCS*, pp. 52–61, 1996.

[20] G. Sivaprasad. Larch/CORBA: Specifying the behavior of CORBA-IDL interfaces. TR 95-27a, Department of Computer Science, Iowa State University, 1995.

[21] O. Sy. *Spécification comportementale de composants CORBA*. PhD thesis, Université de Toulouse I, 2001.

[22] C. Szyperski. *Component Software – Beyong Object Oriented Programming.* Addison-Wesley, 1998.