

# Combining Graphical and Formal Development of Open Distributed Systems\*

Einar B. Johnsen<sup>1,3</sup>, Wenhui Zhang<sup>2</sup>, Olaf Owe<sup>1</sup>, Demissie B. Aredo<sup>2,4</sup>

<sup>1</sup>Dept. of Informatics, University of Oslo, Norway, {einarj, olaf}@ifi.uio.no

<sup>2</sup>Institute for Energy Technology, Halden, Norway

<sup>3</sup>BISS, FB3, University of Bremen, Germany, einar@tzi.de

<sup>4</sup>Norwegian Computing Center, Oslo, Norway, aredo@nr.no

**Abstract** A specification of a software system involves several aspects. Two essential aspects are convenience in specification and possibility for formal analysis. These aspects are, to some extent, exclusive. This paper describes an approach to the specification of systems that emphasizes both aspects, by combining UML with a language for description of the observable behavior of object viewpoints, OUN. Whereas both languages are centered around object-oriented concepts, they are complementary in the sense that one is graphical and semi-formal while the other is textual and formal. The approach is demonstrated by a case study focusing on the specification of an open communication infrastructure.

## 1 Introduction

In order to develop open distributed systems, we need techniques and tools for specification, design, and code generation. For the specification of such systems, it can be desirable to use graphical notations, so that specifications are intuitive and easy to understand, and misunderstandings and mistakes thereby hopefully avoided. On the other hand, it is also desirable for the specification technique to have a formal basis which supports rigorous reasoning about specifications and designs. As there is no single existing method that covers all the desired aspects adequately, we have chosen to extend, adapt, and combine existing formal methods and tools into a platform for specification, design, and refinement of open distributed systems. In this approach, we integrate the Unified Modeling Language (UML) [6, 20] modeling techniques, the Oslo University Notation (OUN) specification language [12, 21], and the Prototype Verification System (PVS) specification language [22] in a common platform [26].

*UML* is a comprehensive notation for creating a visual model of a system. It is a dynamic specification language based on a combination of popular modeling languages [5, 10, 24] and has become a widely used standard for object-oriented software development. As a modeling language, UML allows a description of a

---

\* This work is financed by the Research Council of Norway under the research program for Distributed IT-Systems.

system in great detail at any level of abstraction. UML does not rely on a specific development process, although it facilitates descriptions and development processes that are case-driven, architecture centric, iterative, and incremental. It provides notations needed to define a system with any particular architecture and does so in an object-oriented way. The graphical notation of UML includes class diagrams, object diagrams, use case diagrams, interaction diagrams (including sequence diagrams and collaboration diagrams), statechart diagrams, activity diagrams, component diagrams, and deployment diagrams. These diagrams allows us to describe central *aspects* of the overall system. However, rigorous reasoning in UML is difficult as the language lacks a formal semantics.

*OUN* is a high-level object-oriented design language which supports the development of open distributed systems. The language is designed to enable formal reasoning in a convenient manner. In particular, reasoning control is based on static typing and proofs, and generation of verification conditions is based on static analysis of program or specification units. A system design in *OUN* consists of classes, interfaces, and contracts, however in this paper we will only consider specification by means of interfaces. An object in *OUN* can support a number of interfaces and this number can change dynamically.

In contrast to OCL [27], *OUN* lets us capture aspects of the *observable behavior* of objects in terms of input and output. For open systems, implementation details of components may not be available, but the behavior of a component can be locally determined by its interaction with the environment [1]. *OUN* objects may have internal activity and run in parallel. They communicate asynchronously by means of remote methods calls and exchange object identities. Object interaction is recorded in *communication traces* [8,14,23], so the semantics of the language is trace-based. In *OUN*, the observable behavior of an object can be specified by several interfaces that represent aspects of the object's behavior. An interface is specified syntactically by an alphabet and semantically by predicates on traces. Hence, an interface identifies a set of finite traces reflecting possible communication histories of a component (or object) at different points of time. Requirement specifications of interfaces consider observable behavior in the form of an input/output-driven assumption guarantee paradigm; invariant predicates about output are guaranteed when assumption predicates about input are respected by the communication environment. For these predicates, we may use patterns that describe the traces of the system in a graphical style reminiscent of message sequence charts.

*PVS* is a specification language based on higher-order logic. It has a rich type system including predicate subtypes and dependent types. These features make the language very expressive, but type checking becomes undecidable. In addition, there are type constructors for functions, tuples, records, and abstract data types. *PVS* specifications are organized into theories and modularity and reuse are supported by means of parameterized theories. *PVS* has a powerful verification tool which uses decision procedures for simplifying and discharging proofs, and provides many proof techniques such as induction, term rewriting, backward proofs, forward proofs, and proof by cases, for interactive user intervention.

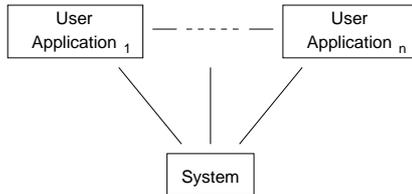
*The purpose of the integration* of the techniques is to exploit the advantages of UML for high-level specification with graphical notations, the formal notation provided by OUN for specification of additional aspects concerning the observable behavior of objects or components, and the theorem-proving capabilities of PVS for verification of correctness requirements. The system development process in our approach consists of the following steps: informal specification of user requirements; partial specifications in UML; extension of the UML interface specifications into OUN specifications; translation of the partial specifications into a PVS specification; verification and validation of the specification with PVS tools; and code generation (for instance towards Java). The emphasis here is on specification, verification aspects are briefly considered at the end of the paper.

This approach is demonstrated by a case study. We consider a specification of the **SoftwareBus** system<sup>1</sup>, an object-oriented data exchange system developed at the OECD Halden Reactor Project [2]. The system itself is not safety critical, but, depending on the user applications of the system, it may have safety implications. For instance, the PLASMA plant safety monitoring and assessment system [7] is based on the **SoftwareBus** communication package. Important functions of the system include: providing the current safety status of the plant, online monitoring of the safety function status trees, displaying the pertinent emergency operating procedures, and displaying the process parameters which are referenced in the procedures. Another application of the **SoftwareBus** system is data communication mechanism in the SCORPIO core surveillance system [16], a system supporting control room operators, reactor physicists, and system supervisors. The SCORPIO system has two modes: monitoring and predicting, and the main purpose of the system is to increase the quality and quantity of information and enhance plant safety by detecting and preventing undesired core conditions. Correctness and reliability of such systems are important, e.g. presentation of wrong information may lead to wrong control actions and trigger safety protection actions, which could contribute to the possibility of failure with safety consequences. Consequently, rigorous specification and formal analysis of the underlying communication framework are important.

The paper is organized as follows. In Section 2, the functionality of the **SoftwareBus** system is described and a possible system architecture proposed. In Section 3, a specification of the **SoftwareBus** system is presented, using the UML modeling techniques. In Section 4, we extend the UML interface specifications into OUN and show how graphical patterns are used to capture the observable behavior of the components. Section 5 considers robustness issues and illustrates how fault-tolerance can be added to the system in OUN. In Section 6, we discuss the usefulness of our approach.

---

<sup>1</sup> As the purpose of this paper is to illustrate our approach to system development, considerable simplifications have been made. Readers interested in the system are referred to the web-page <http://www.ife.no/swbus> for detailed documentation.



**Figure 1.** Basic structure of the **SoftwareBus** system with user applications.

## 2 Functionality of the Software Bus

The main motivation for constructing distributed systems considered in this paper arises from the need for surveillance and control of processes in power plants. For this purpose, data collected from processes have to be processed and presented. As the same set of data may be a basis for presentation in different forms at different locations, data sharing among user applications is a necessity.

To begin with, we may think of a *system* with an unknown number of potential user applications connected to it, as shown in Figure 1. The user applications communicate with the system in order to carry out necessary data processing tasks. These include the creation of variables, the assignment of values to variables, accessing the values of variables, and the destruction of variables.

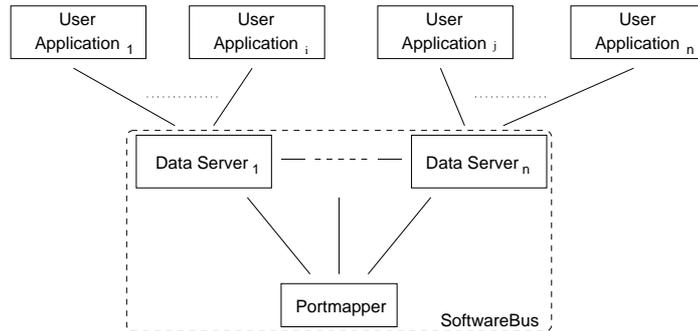
The **SoftwareBus** system is an object-oriented system in which classes, functions, and variables are treated as **SoftwareBus** objects, i.e. as manipulatable units in the **SoftwareBus** system. A **SoftwareBus** object can be identified in two ways. First, by its identifier: In the implementation of the **SoftwareBus** system, an object identifier identifies a row in a table where object information is stored. Second, by the object's name and its parent's identifier, as objects are organized in a hierarchy where the top level represents the local application or proxy objects representing remote applications. A pointer type is used for the contents of objects, i.e. pointers to places in user applications where the contents of objects are stored, and another type is used for codes of functions. In addition, variables holding references to remote applications are grouped as a type. Therefore, we have the following basic types: **SbTName** for object names, **SbTSti** for local object identifiers, **SbTContents** for contents of objects, **SbTCode** for code of functions, and **SbTApplication** for references to applications. **SbTSti** is a general type for identifying **SoftwareBus** objects. We classify different objects or objects used in different contexts into subtypes of **SbTSti**.

### 2.1 System Interfaces

With the system architecture of Figure 1, we only need to consider one interface, which is the interface the system provides towards user applications. This interface includes operations for initializing user applications, establishing logical connections with other applications, and manipulating **SoftwareBus** objects:

<b>Name</b>	sb_initialize
<b>Arguments</b>	name: <b>SbTName</b>
<b>Return Value</b>	none
<b>Description</b>	signals that a calling user application enters the system.
<b>Name</b>	sb_exit
<b>Arguments</b>	none
<b>Return Value</b>	none
<b>Description</b>	signals that the calling user application leaves the system.
<b>Name</b>	sb_connect_appl
<b>Arguments</b>	appl_name: <b>SbTName</b>
<b>Return Value</b>	appl_ref: <b>SbTApplication</b>
<b>Description</b>	establishes a logical connection with <i>appl_name</i> .
<b>Name</b>	sb_disconnect_appl
<b>Arguments</b>	appl_ref: <b>SbTApplication</b>
<b>Return Value</b>	none
<b>Description</b>	destroys the logical connection to <i>appl_ref</i> .
<b>Name</b>	sb_id
<b>Arguments</b>	name: <b>SbTName</b> parent_ref: <b>SbTStiParent</b>
<b>Return Value</b>	obj_ref: <b>SbTSti</b>
<b>Description</b>	obtains the reference of (a proxy of) the object identified by <i>name</i> and <i>parent_ref</i> .
<b>Name</b>	sb_delete_obj
<b>Arguments</b>	obj_ref: <b>SbTSti</b>
<b>Return Value</b>	none
<b>Description</b>	deletes the object identified by <i>obj_ref</i> .

The operations **sb\_initialize** and **sb\_exit** are invoked by a user application in order to enter and leave the **SoftwareBus** system, respectively. The operations **sb\_connect\_appl** and **sb\_disconnect\_appl** concern the logical connections between processes. To establish a logical connection, the remote application (identified by *appl\_name*) must have entered the system prior to the method invocation. The operations **sb\_id** and **sb\_delete\_obj** are for object manipulation. For brevity, other object manipulation operations (e.g. for creating subclasses, modifying classes, creating instances of classes, using attributes and methods of such instances) are not described here. The parameter *parent\_ref* of the operation **sb\_id** identifies either the local application, an object in the local application, a remote application, or a proxy of an object of a remote application. A proxy needs to be created if *name* identifies a remote object without a proxy in the local application. **SbTStiParent** is a subtype of **SbTSti** that represents **SoftwareBus** objects used as parent objects in given contexts.



**Figure 2.** Decomposition of **SoftwareBus**.

## 2.2 Decomposition of the System

A centralized system architecture is not the best choice for a distributed computing environment with respect to efficiency, communication overhead, and reliability. A better solution is to keep as much data as possible in or near the user applications that possess the data, and provide a mechanism for data sharing. Figure 2 shows a decomposition of the system based on this principle.

With this architecture, the system consists of a central unit and a set of data servers. The purpose of the central unit is to maintain information about the data servers and their user applications, while the purpose of the servers is to store data that is shared among user applications. In this system, a user application communicates with a data server in order to carry out necessary data processing tasks. Depending on requests from the user application, the data server may communicate directly with another data server to fulfill the requests or communicate with the central unit if information about other servers is requested or needed. The number of data servers and their locations is not predetermined. Data servers may be started at any location, whenever necessary.

Two interfaces need to be specified. One is the interface of the central unit towards data servers and the other is the interface of a data server to other data servers. To be consistent with the terminology of the **SoftwareBus** documentation [2], we call the central unit a **portmapper** in the sequel. The interface provided by the portmapper to data servers includes the following operations: **pm\_initialize**, **pm\_exit**, **pm\_connect\_appl**, and **pm\_disconnect\_appl**.

These operations are the internal equivalents of the data server operations starting with **sb**, except that the internal operations have an additional input parameter of type **SbTApplication**. Think that all calls from user applications to the system are delivered by a data server. Upon receipt of a call **sb\_m** (with  $m$  being one of **initialize**, **exit**, **connect\_appl**, and **disconnect\_appl**) from a user application, the data server forwards the call to the portmapper by

calling `pm_m`. The additional input parameter is used to identify the calling user application in order to ensure that returns to calls are transmitted correctly.

The operations of the interface provided by data servers to other data servers are similar (with respect to the functionality) to the interface provided by the system to user applications, if we omit the operations associated with the portmapper, i.e. the operations `sb_initialize`, `sb_exit`, `sb_connect_appl`, and `sb_disconnect_appl`. When a user application calls an operation of the system, these four operations are forwarded to the portmapper while the other operations are handled by a data server. The data server may issue a corresponding call to another data server when necessary.

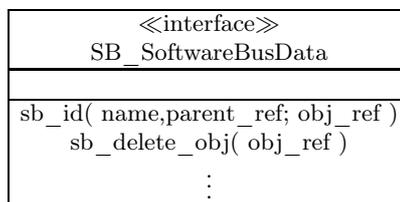
### 3 UML Specification

In this section, we present a specification of the **SoftwareBus** system using UML modeling techniques. First, we give static structural descriptions of the major system components such as interfaces, classes (or objects), and relationships among them, as outlined in the previous section. For this purpose, we model basic elements like classes, components, and the interfaces that they provide to each other. Then, the static structure and the dynamic behavior of the system can be specified by putting together the basic elements into UML diagrams.

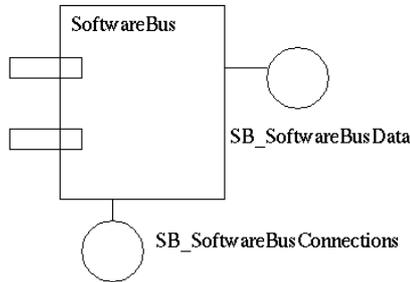
#### 3.1 External Interfaces

The external interface of the **SoftwareBus** specifies operations available to user applications. These operations can be grouped into two: those concerned with object manipulations and those dealing with communication between user applications and the **SoftwareBus** system. Accordingly, we decompose the interface towards external user applications into two subinterfaces: **SB\_SoftwareBusData** and **SB\_SoftwareBusConnections**. The operations of these interfaces have been discussed in the previous section. Here, we describe the interfaces and their relationships to the **SoftwareBus** system using the UML graphical notation.

For the purpose of this paper, the interface **SB\_SoftwareBusData** includes at least the operations `sb_id` and `sb_delete_obj`. In the actual **SoftwareBus** system, several other operations are provided for object manipulation. The specification, given as a UML interface, is as follows.



The interface **SB\_SoftwareBusData** consists of operations for object manipulation. The interpretation of operation signatures is as follows. In the list



**Figure 3.** **SoftwareBus** interfaces.

of parameters, values that occur before the symbol “;” are *input* parameters, whereas the remaining values are *output* parameters. The types of the parameters are as specified in the previous section.

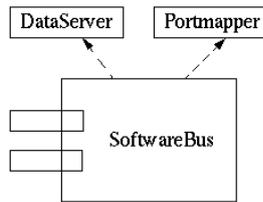
The interface **SB\_SoftwareBusConnections** consists of operations that handle connections between the system and the user applications. The specification, given as a UML interface, is as follows.

<<interface>> <b>SB_SoftwareBusConnections</b>
sb_initialize( name ) sb_exit() sb_connect_appl( appl_name; appl_ref ) sb_disconnect_appl( appl_ref )

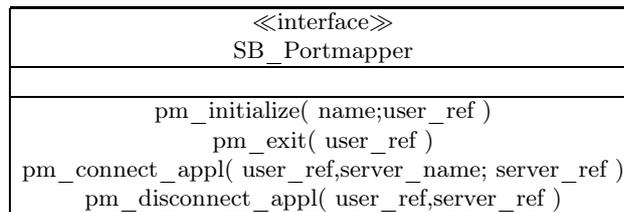
The relationships between the **SoftwareBus** system and its external interfaces are depicted in a UML class diagram in Figure 3.

### 3.2 Internal Interfaces

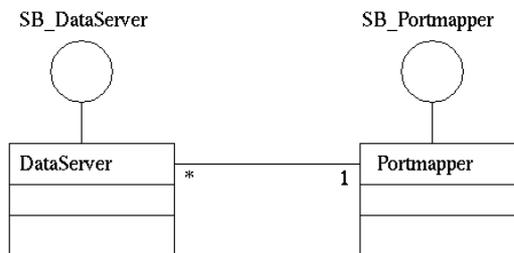
By “internal” interfaces, we mean the interfaces that the different parts of the **SoftwareBus** system provide to each other, in contrast to the interfaces that are available to the user applications. The internal structure of the **SoftwareBus** system consists of a portmapper with a set of data servers. A description of the **SoftwareBus** system in terms of its classes is shown in Figure 4. We consider two interfaces **SB\_Portmapper** and **SB\_DataServer**, provided by the portmapper to data servers and by data servers to other data servers, respectively. **SB\_Portmapper** resembles **SB\_SoftwareBusConnections**, specified as follows:



**Figure 4.** The **SoftwareBus** component and its classes.



The interface **SB\_DataServer** is similar to **SB\_SoftwareBusData**, so the description is omitted. The class diagram given in Figure 5 shows the classes and internal interfaces of the **SoftwareBus** system and relationships among them.



**Figure 5.** **SoftwareBus** class diagram.

*Remarks on Implementation Issues.* In theory, a user application may communicate with the system by contacting any data server. However, this complicates the communication pattern. A possible solution is a one-to-one mapping between user applications and data servers. With such a mapping, parameters to method calls can be simplified, as the identity of the user application at the source of a call is given by its corresponding data server. In this paper, we have adopted a more general solution, where several applications can use the same data server.

For simplicity, we nevertheless assume that an application does not employ several data servers during the same session (i.e. without exiting the **SoftwareBus** and reconnecting via the new server). In practice, a user application and its corresponding data server are often implemented in one component [2].

## 4 OUN Specification

OUN allows the specification of observable behavior by means of interfaces. Thus, an object considered through an interface represents a specific viewpoint to the services provided by the object. There may be several interfaces associated with an object, which give rise to supplementary behavioral specifications of the same object. Proof obligations arise in order to verify that objects of a given class actually fulfill the requirements of the interfaces they claim to implement [21]. In this section, we semantically restrict the UML interfaces of Section 3 to obtain OUN interfaces that express the *behavior* of the **SoftwareBus** interfaces.

In OUN, the interfaces of an object do not only contain syntactic method declarations, they also specify aspects of the observable behavior of that object, i.e. the possible communication histories of the object when a particular subset of its alphabet is taken into account. The alphabet of an interface consists of a set of communication events reflecting the relevant method calls for the currently considered role of the object. Say that a method “m” is implemented by some object  $o$  and it is called by another object  $o'$  with parameters  $p_1, \dots, p_n$  and returns with values  $v_1, \dots, v_m$ . The call is represented by two distinct communication events: first,  $o' \rightarrow o.m(p_1, \dots, p_n)$  reflects the *initiation* of the method call and then,  $o' \leftarrow o.m(p_1, \dots, p_n; v_1, \dots, v_m)$  reflects the *completion* of the call.

The aspect of an object’s behavior that is specified by an interface is given by (first-order) predicates on finite sequences of such communication events. For each interface there are two (optional) predicates, an assumption and an invariant. The assumption states conditions on objects in the environment of the object, so it is a predicate that should hold for sequences that end with input events to the object. The assumption predicate should be respected by every object in the environment; we consider communication with one object at a time. The invariant guarantees a certain behavior when the assumption holds, so the invariant is given by a predicate on the sequences ending with output from the object. When an assumption or an invariant predicate is omitted in the specification of an interface, we understand it as “true”.

For the specification of the boundary between users and the **SoftwareBus** system, two interfaces are introduced: **SB\_SoftwareBus** and **SB\_User**. The first interface is offered by the **SoftwareBus** to its users. The second interface is an “empty” interface used to identify users. OUN objects are always considered through interfaces, so all objects need interfaces, even if they do not contain methods that are accessible to the environment. In the next section, we consider requirements on the OUN objects that provide the **SB\_SoftwareBus** interface.

#### 4.1 External Interfaces

In OUN, an interface may inherit other interfaces. Using inheritance, the interface of the **SoftwareBus** can be specified as follows.

```

interface SB_SoftwareBus
  inherits SB_SoftwareBusData,
           SB_SoftwareBusConnections
begin
end
```

Here, *SB\_SoftwareBusData* and *SB\_SoftwareBusConnections* are as discussed in the previous section, without any semantic restrictions. Operations are inherited automatically from the UML interface specifications via the Integrator [26]. In this section, we extend the interfaces with behavioral constraints.

*SB\_SoftwareBusData*. This interface considers object manipulation between applications in the **SoftwareBus**. (For remote object manipulation, logical connections between applications are established via the *SB\_SoftwareBusConnections* interface.) After initialization, the user application may talk to the **SoftwareBus** in order to create new objects and manipulate existing objects. It is assumed that no objects are known a priori to a user. References to objects are obtained from the **SoftwareBus** before they are passed on as arguments in method invocations. Knowledge of an object reference is obtained either through the creation of that object by the user or by a call to *sb\_id* with the appropriate parameters. We capture knowledge of object references by a predicate on the history. Let *known(x, y, r, h)* express that the object reference *r* is known by the application *x*, where *h* is its history of communication with the **SoftwareBus** *y*, defined as follows for the operations considered in this paper:

$$\begin{aligned}
 \textit{known}(x, y, r, \varepsilon) &= \mathbf{false} \\
 \textit{known}(x, y, r, h \vdash x \leftarrow y.\textit{sb\_id}(\_, \_ ; r)) &= \mathbf{true} \\
 \textit{known}(x, y, r, h \vdash x \leftarrow y.\textit{sb\_delete\_obj}(r)) &= \mathbf{false} \\
 \textit{known}(x, y, r, h \vdash \textit{others}) &= \textit{known}(x, y, r, h)
 \end{aligned}$$

Here, cases are considered in the order listed (à la ML), so we recursively inspect the history until one of the cases apply. The symbols “ $\varepsilon$ ” and “ $\vdash$ ” denote the empty trace and the *right append* operation, respectively. Furthermore, we represent by “ $\_$ ” an uninteresting parameter in parameter lists. In the last case, “*others*” will match any event. When the history is empty, the object reference has not been obtained and the predicate returns “false”. Likewise, if an object has been deleted, it cannot have a reference. If the reference has been obtained by means of the method *sb\_id*, the predicate returns “true”. As we consider the last event first, we normally do not need to inspect the entire history.

The communication environment of the **SoftwareBus** is dynamic as objects may be (remotely) created and destroyed at run-time. We want to capture by a function on the history the requirement that objects are referred to only when they are known to the user application. If a method call refers to *i* objects, the

references to all  $i$  objects must be checked. (In this simplified version of the **SoftwareBus**, we only consider methods with one reference to check.) Define a predicate  $correctObjRef(x, y, h)$  to express that all object references that are passed as parameters to events between the user application  $x$  and the **SoftwareBus**  $y$  in the history  $h$  are known to  $x$ , as follows:

$$\begin{aligned}
correctObjRef(x, y, \varepsilon) &= \mathbf{true} \\
correctObjRef(x, y, h \vdash x \rightarrow y.sb\_id(\_, r; \_)) &= known(x, y, r, h) \wedge correctObjRef(x, y, h) \\
correctObjRef(x, y, h \vdash x \rightarrow y.sb\_delete\_obj(r)) &= known(x, y, r, h) \wedge correctObjRef(x, y, h) \\
correctObjRef(x, y, h \vdash \text{others}) &= correctObjRef(x, y, h)
\end{aligned}$$

The methods of SB\_SoftwareBusData should only be available to users, i.e. access should be restricted to objects that provide the SB\_User interface, which is an empty interface in the sense that it contains no methods. This is done by a WITH-clause in the specification of SB\_SoftwareBusData. In the interface below, we denote by “this” the object specified by the interface and by “caller” an object offering the SB\_User interface.

```

interface SB_SoftwareBusData
begin
  with SB_User
    [operations as in the UML specification]
  asm  $correctObjRef(\text{caller}, \text{this}, h)$ 
end

```

In the specification, **asm** is the keyword preceding assumptions. Assumptions are requirements on the environment, i.e. they are expected to hold for histories ending with input to “this” object. Inputs to an object  $x$  are either initiations of calls to methods declared in  $x$  or completions of calls made by  $x$  to other objects. It is implicit in the formalism that assumption predicates must hold for output as well as input, as the trace sets of interfaces are prefix-closed [11]. An object may not break its own assumption. Although no particular invariant predicate is specified for SB\_SoftwareBusData, we still get as invariant the assumption predicate, but for histories that end with output from “this” object.

*SB\_SoftwareBusConnections.* The sequence of events expected to hold between the **SoftwareBus** and a particular application is described by an assumption, naturally expressed by a prefix of a pattern.

Let  $p$  be an OUN object which offers the SB\_SoftwareBusConnections interface (i.e.  $p$  is the **SoftwareBus**). Let  $h$  be the history of  $p$ . The sequence of events in the alphabet of the **SoftwareBus** that reflect calls from some user application  $a$  can be described by the following predicate (for convenience, we write  $c\_a$  for  $connect\_appl$  and  $d\_a$  for  $disconnect\_appl$ ).

$$\begin{aligned}
correctComSeq(a, p, h) &= \\
h \mathbf{prp} [a \leftrightarrow p.sb\_initialize() [a \leftrightarrow p.sb\_c\_a(\_; \_) a \leftrightarrow p.sb\_d\_a(\_)]^* & \\
a \leftrightarrow p.sb\_exit()]^* &
\end{aligned}$$

In the specifications,  $h \text{ ptn } P$  denotes that the trace  $h$  adheres to the pattern  $P$ , which is a regular expression extended with simple pattern matching [11]. Subpatterns may be enclosed in square brackets. The prefix predicate  $h \text{ prp } P$  expresses *invariant properties*; it is true if there is an extension  $h'$  of  $h$  such that  $hh' \text{ ptn } P$ , where  $hh'$  is the concatenation of the two traces  $h$  and  $h'$ . The notation  $o_1 \leftrightarrow o_2.m(\dots)$  is shorthand for the initiation event  $o_1 \rightarrow o_2.m(\dots)$  immediately succeeded by the completion  $o_1 \leftarrow o_2.m(\dots)$ . This corresponds to synchrony if the two events are perceived as immediately succeeding each other by both parties. For brevity, the same notation used in sets represents both the initiation and the completion event. Using projection on the history, the more graphical style of behavioral constraints given by patterns can also be used to capture specific aspects of the behavior, rather than the more state-resembling predicates previously encountered. Denote by  $h/S$  a trace  $h$  restricted to events of a set  $S$ . The following predicate determines whether an application  $a$  is registered in the **SoftwareBus** system with portmapper  $p$  after history  $h$ :

$$\begin{aligned} up(a, p, h) = & \\ & h/\{a \leftrightarrow p.sb\_initialize(), a \leftrightarrow p.sb\_exit()\} \\ & \text{ptn } [a \leftrightarrow p.sb\_initialize() \ a \leftrightarrow p.sb\_exit()]* \ a \leftrightarrow p.sb\_initialize(). \end{aligned}$$

The **SoftwareBus** may receive requests from an application  $a_1$  for the reference of another application  $a_2$  (known to  $a_1$  only by name) in order to establish a logical connection. Before such a connection is possible, both  $a_1$  and  $a_2$  must already be registered in the system. The following predicate determines whether  $a_1$  has a logical connection to  $a_2$ :

$$\begin{aligned} connection(a_1, a_2, p, h) = & \\ & h/\{a_1 \leftrightarrow p.sb\_c\_a(\_ ; a_2), a_1 \leftrightarrow p.sb\_d\_a(a_2)\} \\ & \text{ptn } [a_1 \leftrightarrow p.sb\_c\_a(\_ ; a_2) \ a_1 \leftrightarrow p.sb\_d\_a(a_2)]* \ a_1 \leftrightarrow p.sb\_c\_a(\_ ; a_2). \end{aligned}$$

Using these predicates, the requirement that connections are only opened to applications that have registered with the **SoftwareBus** and that only those connections are closed that are currently open, is expressed by a predicate on the history, checking that  $up$  holds before  $sb\_c\_a(a_2)$  is called and that  $connection$  holds before  $sb\_d\_a(a_2)$  is called.

$$\begin{aligned} cn(p, \varepsilon) &= \text{true} \\ cn(p, h \vdash a_1 \leftrightarrow p.sb\_c\_a(\_ ; a_2)) &= cn(p, h) \wedge up(a_1, p, h) \wedge up(a_2, p, h) \\ cn(p, h \vdash a_1 \leftrightarrow p.sb\_d\_a(a_2)) &= cn(p, h) \wedge connection(a_1, a_2, p, h) \\ cn(p, h \vdash \text{others}) &= cn(p, h) \end{aligned}$$

With these predicates, the interface `SB_SoftwareBusConnections` can be specified as follows:

```

interface SB_SoftwareBusConnections
begin
  with SB_User
    [operations as in the UML specification]
  asm correctComSeq(caller, this, h)
  inv cn(this, h)
end
```

In the specification, **inv** is the keyword preceding invariant predicates. Invariants are guaranteed by the object offering the interface, provided that the assumption is not broken by *any* object in the environment. Invariant predicates are consequently expected to hold for histories ending with output from “this” object, in contrast to the assumptions that should hold for histories ending with input. Thus, interface behavior is specified following an input/output-driven assumption guarantee paradigm.

## 4.2 Internal Interfaces

As explained in Section 3, the internal structure of the **SoftwareBus** consists of a portmapper with a set of data servers. We now consider interfaces for the portmapper and the data servers in OUN.

The interface **SB\_Portmapper** is similar to **SB\_SoftwareBusConnections**. Let  $p$  be an OUN object which offers the **SB\_Portmapper** interface (i.e.  $p$  is the portmapper). Let  $h$  be the history of  $p$ . The sequence of calls that we expect from a data server  $d$  to the portmapper  $p$  can be described by a predicate:

$$\begin{aligned} \text{correctComSeq}'(d, p, h) = \\ h \text{ prp } [d \leftrightarrow p.\text{pm\_initialize}(\_; \_) [d \leftrightarrow p.\text{pm\_c\_a}(\_, \_; \_) d \leftrightarrow p.\text{pm\_d\_a}(\_, \_)]^* \\ d \leftrightarrow p.\text{pm\_exit}(\_)]^* \end{aligned}$$

The following predicates on the history of the portmapper are used to determine whether an application  $a$  is registered in the **SoftwareBus** system and whether the application  $a_1$  has a logical connection to  $a_2$ , respectively.

$$\begin{aligned} \text{up}'(a, p, h) = \\ \exists d : h / \{d \leftrightarrow p.\text{pm\_initialize}(n; a), d \leftrightarrow p.\text{pm\_exit}(a)\} \\ \text{ptn } [d \leftrightarrow p.\text{pm\_initialize}(n; a) d \leftrightarrow p.\text{pm\_exit}(a)]^* d \leftrightarrow p.\text{pm\_initialize}(n; a). \end{aligned}$$

$$\begin{aligned} \text{connection}'(a_1, a_2, p, h) = \\ \exists d : h / \{d \leftrightarrow p.\text{pm\_c\_a}(a_1, \_; a_2), d \leftrightarrow p.\text{pm\_d\_a}(a_1, a_2)\} \\ \text{ptn } [d \leftrightarrow p.\text{pm\_c\_a}(a_1, \_; a_2) d \leftrightarrow p.\text{pm\_d\_a}(a_1, a_2)]^* d \leftrightarrow p.\text{pm\_c\_a}(a_1, \_; a_2). \end{aligned}$$

In the **SoftwareBus** system, applications connect and disconnect to each other. Two applications should not attempt to connect unless both are registered with the portmapper. Also, two applications should not attempt to disconnect unless they already have an open connection. This can be expressed by a predicate on the history of the portmapper as follows.

$$\begin{aligned} \text{cn}'(p, \varepsilon) &= \text{true} \\ \text{cn}'(p, h \vdash d \leftrightarrow p.\text{pm\_c\_a}(a_1, \_; a_2)) &= \text{cn}'(p, h) \wedge \text{up}'(a_1, p, h) \wedge \text{up}'(a_2, p, h) \\ \text{cn}'(p, h \vdash d \leftrightarrow p.\text{pm\_d\_a}(a_1, a_2)) &= \text{cn}'(p, h) \wedge \text{connection}'(a_1, a_2, p, h) \\ \text{cn}'(p, h \vdash \text{others}) &= \text{cn}'(p, h) \end{aligned}$$

The expected behavior of the data servers in the environment becomes the assumption predicate of the `SB_Portmapper` interface. The correct transmittal of references is the responsibility of the portmapper. Hence, using the predicates defined above, the interface `SB_Portmapper` can be specified as follows:

```

interface SB_Portmapper
begin
  with SB_DataServer
    [operations as in the UML specification]
  asm correctComSeq'(caller,this, h)
  inv cn'(this, h)
end

```

`SB_DataServer` is similar to `SB_SoftwareBusData` and is omitted here.

## 5 Adding Robustness to the Portmapper

The **SoftwareBus** system, as specified in the previous sections, is clearly prone to errors. In particular, the **SoftwareBus** has a dynamic structure where user applications can join or exit the system at any time and objects can be (remotely) created, modified, and deleted. This may cause difficulty as previously valid object or application names and references may no longer be available to the environment. Of primary interest is the robustness of the portmapper, on which the entire system depends. We want to remove situations that may cause deadlock in the portmapper, to ensure that the communication framework is operative even when user applications have deadlocked. (In open distributed systems like the **SoftwareBus**, the robustness of user applications is outside our control.) Possible deadlocks in the portmapper will be avoided by issuing *exceptions* in response to method calls from user applications when regular behavior is out of place. In this section, we consider modifications of the specifications that make the portmapper more robust.

We will follow the methodology outlined in [13], where several refinement relations are proposed for a stepwise development of OUN specifications with regard to fault-tolerance. The relations ensure that, after appropriate manipulation of the traces, the behavior of the intolerant specification is recovered. The exact relation required in each case depends on how the safety and liveness properties of the intolerant specification should be preserved. (In this paper, only safety properties are considered.) The occurrence of a fault is represented in the formalism by a special event, replacing the usual completion event and thus acting as an error message in response to a method call. Upon receipt of the error message, the calling object may choose its course of action. In the syntax of OUN interfaces, the keyword **throws** precedes the fault classes that are handled by an exceptional completion event.

In the specification of the portmapper, consider the following exceptions:

- *F1*: An application tries to register to the system with a name that is in use.
- *F2*: An application tries to connect to another, unavailable application.

- *F3*: An application tries to close a connection that is not open.

We refer to these error situations as fault classes *F1*, *F2*, and *F3*, respectively. In all three cases, the portmapper should return an exceptional completion event in response to the erroneous method call and continue as if the fault had not occurred. We now modify the interface of the portmapper accordingly.

Inside the **SoftwareBus**, data servers register user applications with the portmapper. Let  $in\_use(h)$  denote a function on the history of the portmapper that calculates the set of application names currently in use for **SoftwareBus** applications. A fault of class *F1* occurs when an application name  $a$  is already in use by a user application at the invocation of  $pm\_initialize(a)$ . If  $x \leftarrow y.m(i_1, \dots, i_n; \dots)$  is a completion event in response to an invocation of a method  $m$  with inputs  $i_1, \dots, i_n$  and  $F$  is a fault class, let  $x \leftarrow y.m_F(i_1, \dots, i_n)$  denote the  $F$ -exception event raised in response to this invocation of  $m$ . Let  $reg(h)$  express that *F1* exception events are issued correctly, where  $h$  is the history of the portmapper:

$$\begin{aligned} reg(h \vdash d \leftarrow p.pm\_initialize(a; \_)) &= a \notin in\_use(h) \wedge reg(h) \\ reg(h \vdash d \leftarrow p.pm\_initialize_{F1}(a)) &= a \in in\_use(h) \wedge reg(h) \\ reg(h \vdash others) &= reg(h) \end{aligned}$$

Here, different completion events are issued in response to calls to  $pm\_initialize(a)$ , depending on whether the name  $a$  is currently used in the system or not.

Next, we specify when exception events for fault classes *F2* and *F3* should be issued. If  $a_1$  attempts to connect to  $a_2$ , the portmapper should respond by an *F2* exception when  $\neg up(a_2, h)$  (where  $h$  is the current history). Similarly, an *F3* exception should be issued in response to an attempt to disconnect a non-existing connection. Let  $cn''$  modify the  $cn'$  predicate:

$$\begin{aligned} cn''(p, \varepsilon) &= \mathbf{true} \\ cn''(p, h \vdash a_1 \leftarrow p.pm\_c\_a(\_, a_2; \_)) &= up(a_2, h) \wedge cn''(p, h) \\ cn''(p, h \vdash a_1 \leftarrow p.pm\_c\_a_{F2}(\_, a_2; \_)) &= \neg up(a_2, h) \wedge cn''(p, h) \\ cn''(p, h \vdash a_1 \leftarrow p.pm\_d\_a(\_, a_2)) &= connection'(a_1, a_2, p, h) \wedge cn''(p, h) \\ cn''(p, h \vdash a_1 \leftarrow p.pm\_d\_a_{F3}(\_, a_2)) &= \neg connection'(a_1, a_2, p, h) \wedge cn''(p, h) \\ cn''(p, h \vdash others) &= cn''(p, h) \end{aligned}$$

The predicates  $cn''$  and  $reg$  regulate *output* from the portmapper, so together they become the invariant of the robust portmapper interface. We will now adapt the assumption of **SB\_Portmapper** to handle exceptional completion events. We assume that for initialization and for opening connections, an application can attempt to repeat a call that fails. Define the new assumption predicate:

$$\begin{aligned} correctComSeq''(a, p, h) = \\ h \text{ prp } [a \leftarrow p.pm\_initialize_{F1}(\_, \_)* a \leftarrow p.pm\_initialize(\_, \_) \\ [a \leftarrow p.pm\_c\_a_{F2}(\_, \_, \_)* a \leftarrow p.pm\_c\_a(\_, \_, \_) \\ a \leftarrow p.pm\_d\_a(\_, \_) a \leftarrow p.pm\_d\_a_{F3}(\_, \_)*] * \\ a \leftarrow p.pm\_exit()*] \end{aligned}$$

With these modified predicates, we specify the robust portmapper interface:

```

interface SB_RobustPortmapper
begin
  with SB_DataServer
    opr pm_initialize(; object_ref ) throws F1
    opr pm_exit( )
    opr pm_c_a( user_ref, server_name; server_ref ) throws F2
    opr pm_d_a( user_ref, server_ref ) throws F3
    asm correctComSeq"(caller, this, h)
    inv cn"(this, h)  $\wedge$  reg(h)
end

```

Ignoring failed method calls in the **SB\_RobustPortmapper** interface, i.e. exception events and the initiation events that correspond to them, we obtain the intolerant interface **SB\_Portmapper**. In this sense, the robust portmapper interface is a fault-tolerant refinement of the intolerant portmapper [13]. If we adapt the data servers to the considered fault classes, faults can to some degree be kept within the **SoftwareBus**, without penetrating to the outside, which gives us a certain fault transparency for the **SoftwareBus** system. The robust portmapper tolerates possibly incorrect and deadlocked user applications and the communication framework per se remains operational.

## 6 Discussion

In this paper, we consider how to combine graphical and formal specification notations to develop a dynamic communication framework. We study a simplified version of the OECD **SoftwareBus** system [2], focusing on the basic infrastructure needed to exchange data between applications. Our study does not cover all the operations and functionality needed for actual applications. The OECD **SoftwareBus** provides this communication functionality using specially designed methods for remote creation, modification, and deletion of objects and classes, which can be captured with the methodology outlined in this paper.

Our approach starts with graphical specifications in UML and we develop a formal specification of the **SoftwareBus** system, concentrating on the central aspects of communication and openness. UML interface specifications are extended into the textual specification language OUN by predicates on traces, in order to enable formal reasoning about behavior. OUN interface behavior relies on explicit communication histories rather than state variables. We illustrate various ways of extracting information from the traces by predicates and patterns that mimic a graphical specification style. In the case study, OUN is used to capture dynamic system behavior, such as initialization of new user applications, connections between applications that open or close, and applications that exit the **SoftwareBus** system, by means of these predicates. Hence, the formalism lets us capture the flexibility of the **SoftwareBus** system in a rigorous way.

OUN supports formal development of specifications, syntactically by means of interface inheritance and semantically by means of refinement. In the case study, more complete specifications of the object interfaces can be obtained

through (multiple) inheritance [12]. With our design, the data servers are of particular interest, forming the junction between the external and internal views of the system. The data servers communicate with local user applications, the portmapper, and with remote user applications connected to the **SoftwareBus**. Through inheritance, we can combine behaviors that are captured in different interfaces. Also, OUN provides us with an incremental approach to system robustness [13], as illustrated in the case study.

The development process proposed here is based on a formalization of UML specifications, rather than a formalization of UML itself. The advantage of using UML constructs for specification is that these constructs are intuitive, commonly accepted, and used in industrial software development. UML constructs are important to describe initial software requirements, normally a result of discussions between users and systems analysts (or software engineers). Extending UML interface specifications, we obtain specifications in OUN that capture dynamic aspects not easily expressible in UML or OCL [27]. Although OCL extends UML with object invariants and pre- and postconditions for methods, precise specification of output from objects is still difficult to capture in OCL as the language expresses constraints on object attributes and not on communication events [15]. OCL, as well as other object-oriented formalisms such as Object-Z [25], Maude [18], and temporal logic-based approaches such as TLA [17], specify components by an abstract implementation using state variables. For open distributed systems, it is perhaps better to perceive the behavior of a component as locally determined by its interaction with the environment [1]. In OUN, interface specification is based on the observable behavior of black-box components.

We are recommended to specify open distributed systems by means of object orientation and multiple viewpoints [9]. Both UML and OUN support these features. OUN is object-oriented, including notions of inheritance and object identity, in contrast to process algebras like CSP [8], the  $\pi$ -calculus [19], LOTOS [4], and Actors [1]. As UML and OCL lack a formally defined semantics and proof system, OUN is a suitable supplement for formal reasoning, with precise notions of composition and refinement.

Our approach relies on PVS [22] as a tool for consistency checking and verification of specifications. Basic modeling constructs and conditions of UML class diagrams can be expressed in the PVS specification language in terms of functions and abstract data types [3]. For further system development, OUN specifications are translated into PVS in order to formally verify the correctness of development steps. As OUN notions of composition and refinement are expressed in PVS, OUN syntax can be translated into PVS in terms of trace sets to take advantage of the PVS theorem proving facilities [11]. A framework for the consistency check was described in [26] where software specification is done within a system development environment which integrates Rational Rose (a tool supporting UML from Rational Software Corporation) and the PVS toolkit in order to cover the software development process from specification of system requirements, system design, and verification, to code generation. Code generation facilities for OUN specifications are currently under development.

## 7 Conclusion

This paper proposes an approach to the specification of open distributed systems, based on a combination of UML and OUN specification techniques. To illustrate the approach, we develop parts of a specification of an open communication framework where data and resources are exchanged between applications and where applications connect and disconnect over time. In the development process, graphical UML constructs are used to specify interfaces, classes, and their relations. Class diagrams in UML are expanded into OUN interface specifications by restricting the implicit history variables of communication calls. With first-order predicates for assumption and invariant clauses, we specify the observable behavior of components. OUN permits formal reasoning about system development and captures certain forms of openness by textual analysis, as shown in the case study.

For the development of open distributed systems, OUN is well-suited as a complement to UML. UML is the de facto industry standard and uses intuitive graphical notation, but it lacks the formalization necessary for rigorous system development and the concepts needed to capture the dynamic aspects of reactive systems. In OUN, reasoning is both compositional and incremental. Software units can be written, formally analyzed, and modified independently, while we have control of the maintenance of earlier proven results. The formalism lets us capture dynamic behavior that is not easily handled in UML.

ACKNOWLEDGMENTS: This work is a part of the ADAPT-FT project. The authors thank H. Jokstad and E. Munthe-Kaas for discussions, suggestions, and helpful comments.

## References

1. G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, Jan. 1997.
2. T. Akerbæk and M. Louka. The software bus, an object-oriented data exchange system. Technical Report HWR-446, OECD Halden Reactor Project, Institute for Energy Technology, Norway, Apr. 1996.
3. D. B. Aredo, I. Traore, and K. Stølen. Towards a formalization of UML class structure in PVS. Research Report 272, Dept. of Informatics, Univ. of Oslo, 1999.
4. T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.
5. G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin-Cummings, Redwood City, CA, 1991.
6. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, Massachusetts, USA, 1999.
7. J. Eiler. Critical safety function monitoring: an example of information integration. In *Proceedings of the Specialists' Meeting on Integrated Information Presentation in Control Rooms and Technical Offices at Nuclear Power Plants (IAEA-I2-SP-384.38)*, pages 123–133, Stockholm, Sweden, May 2000.

8. C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice Hall, Englewood Cliffs, NJ, 1985.
9. ITU Recommendation X.901-904 ISO/IEC 10746. *Open Distributed Processing - Reference Model parts 1-4*. ISO/IEC, July 1995.
10. I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.
11. E. B. Johnsen and O. Owe. A PVS proof environment for OUN. Research Report 295, Department of informatics, University of Oslo, 2001.
12. E. B. Johnsen and O. Owe. A compositional formalism for object viewpoints. In *Proc. 5th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2002)*. Kluwer Academic, 2002. To appear.
13. E. B. Johnsen, O. Owe, E. Munthe-Kaas, and J. Vain. Incremental fault-tolerant design in an object-oriented setting. In *Proceedings of the Asian Pacific Conference on Quality Software (APAQS'01)*. IEEE press, Dec. 2001.
14. G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing 74: Proceedings of the IFIP Congress 74*, pages 471–475. IFIP, North-Holland Publishing Co., Aug. 1974.
15. A. Kleppe and J. Warmer. Extending OCL to include actions. In A. Evans, S. Kent, and B. Selic, editors, *Proceedings of UML 2000*, volume 1939 of *Lecture Notes in Computer Science*, pages 440–450, York, UK, Oct. 2000. Springer-Verlag.
16. F. Kostihä. Establishment of an on-site infrastructure to facilitate integration of software applications at Dukovany NPP. In *Proceedings of the Specialists' Meeting on Integrated Information Presentation in Control Rooms and Technical Offices at Nuclear Power Plants (IAEA-I2-SP-384.38)*, Stockholm, Sweden, May 2000.
17. L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
18. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
19. R. Milner. *Communicating and Mobile Systems: the  $\pi$ -Calculus*. Cambridge University Press, May 1999.
20. Object Management Group. *UML Language Specification, version 1.4*, Sept. 2001.
21. O. Owe and I. Ryl. OUN: a formalism for open, object oriented, distributed systems. Research Report 270, Dept. of informatics, Univ. of Oslo, Aug. 1999.
22. 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, Feb. 1995.
23. D. L. Parnas and Y. Wang. The trace assertion method of module interface specification. Technical Report 89-261, Department of Computing and Information Science, Queen's University at Kingston, Kingston, Ontario, Canada, Oct. 1989.
24. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.
25. G. Smith. *The Object-Z Specification Language*. Advances in Formal Methods. Kluwer Academic Publishers, 2000.
26. I. Traore, D. B. Aredo, and K. Stølen. Formal development of open distributed systems: Towards an integrated framework. In *Proc. Workshop on Object-Oriented Specification Techniques for Distributed Systems and Behaviours (OOSDS'99)*, Paris, France, Sept. 1999.
27. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Object Technology Series. Addison-Wesley, 1999.