University of Oslo
Department of Informatics

Combining Graphical and Formal Specification: the Software Bus Case Study

Einar B. Johnsen,
Wenhui Zhang,
Olaf Owe,
Demissie B. Aredo

Research report 297
ISBN 82-7368-247-1

October 2001

Combining Graphical and Formal Specification: the Software Bus Case Study*

Einar B. Johnsen[1] Wenhui Zhang[2]
Olaf Owe[1] Demissie B. Aredo[2]

[1]Department of informatics, University of Oslo, Norway,
email: {einarj, olaf}@ifi.uio.no
[2]Institute for Energy Technology, Halden, Norway,
email: {wenhuiz, demissie}@hrp.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. In this paper, emphasis is placed on specification aspects. Verification aspects are discussed briefly at the end of the paper.

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 is desirable to use graphical notations, so that specifications can easily be understood, in order to avoid misunderstandings and mistakes. On the other hand, it is desirable for the specification technique (or an important part of the technique) to have a formal basis, in order to support rigorous reasoning about specifications and designs. As there is no single existing method that covers all the desired aspects, 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 UML (Unified Modeling Language) [7, 26] modeling techniques, the OUN (Oslo University Notation) specification language [15, 27], and the PVS (Prototype Verification System) specification language [8, 28] in a common platform.

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

The Unified Modeling Language is a comprehensive notation for creating a visual model of a system. It is a dynamic specification language based on a combination of the popular modeling languages from Grady Booch [6], Jim Rumbaugh [30], and Ivar Jacobsen [12] and has become a widely used standard for object-oriented software development. As a modeling language, the UML allows a description of a 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 vocabulary of UML encompasses three kinds of building blocks: things, relationships, and diagrams. The graphical notation 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. However, rigorous reasoning in UML is difficult as the language lacks a formal semantics.

OUN is a high-level object-oriented design language developed at the University of Oslo. The main objective of the language is to support 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. An OUN specification consists of classes, interfaces, and contracts, however in this paper we will only consider specification by means of interfaces. An object may support a number of interfaces and this number may change dynamically.

In contrast to OCL [34], OUN can be used to specify aspects of the *observable behavior* of objects. For open systems, we do not have local control of the components and implementation details of components are generally unavailable. Instead, the behavior of a component can be locally determined by its interaction with the environment [1]. Such interaction between a component and its environment can be recorded in a *communication trace* [10, 19, 29]. In OUN, the external behavior of an object can be specified by several interfaces, each representing an aspect of the object's behavior. The observable behavior of an interface is specified syntactically by an alphabet and semantically by predicates.

OUN objects may have internal activity and run in parallel. They communicate asynchronously by means of remote methods calls and may exchange object identities. The semantics of the language is based on traces, i.e. finite communication histories. A specification is a set of traces reflecting the possible, different communication histories of the 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 graphical patterns that describe the traces of the system.

The PVS specification language is based on higher-order logic. The language has an associated verification system which is implemented in Common Lisp and available from SRI International Computer Science Laboratory. It has a rich type system including the notion of 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. The language was designed to describe computer systems, concentrating on abstract descriptions. 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 the specification aspects. Verification aspects are discussed briefly at the end of this paper.

This approach is demonstrated by a case study. We consider a specification of the **SoftwareBus** system[1], an object-oriented data exchange system developed at the OECD Halden Reactor Project [2]. (Readers interested in the system are referred to the web-page `http://www.ife.no/swbus` for detailed documentation.) The system itself is not safety critical, however, depending on the kind of user applications of the system, it may have safety implications. For instance, the PLASMA plant safety monitoring and assessment system [9] 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 [21], 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 possibility for 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 is discussed. 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, demonstrating how graphical patterns are used

[1] As the purpose of this paper is to illustrate our approach to system development, simplifications have been made and the specification presented in the following sections may not necessarily match all actual requirements and behaviors of the **SoftwareBus** system.

to capture the observable behavior of the components. Section 5 considers robustness issues and illustrates how incremental fault-tolerance can be achieved using OUN. In Section 6, we discuss the usefulness of our approach. A short version of this paper appears in the proceedings of the Asian Pacific Software Engineering Conference (APSEC 2001) [18].

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 surveillance and control of processes, 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 different 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. 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 basic structure of the system with user applications is illustrated in Figure 1.

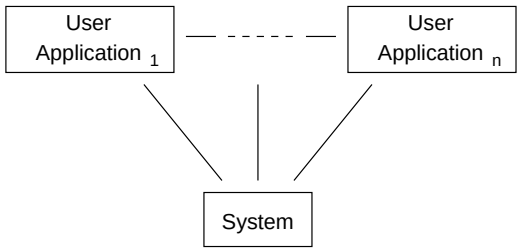


Figure 1: The **SoftwareBus** system with user applications.

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.

For each **SoftwareBus** object, there are two ways to identify the object. One is by its identifier. In the implementation of the **SoftwareBus** system, an object identifier is a pointer to a table where information about the object is stored. The other is by the object's name and its parent's identifier, as objects are organized in a hierarchy where the top level is a constant representing 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	Names of objects
SbTSti	Local identifiers of objects
SbTContents	Contents of objects
SbTCode	Code of functions
SbTApplication	References to applications

The type **SbTSti** is a general type for identifying **SoftwareBus** objects. We may 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:

Name	sb_initialize
Arguments	name: **SbTName**
Return Value	none
Description	signals that a calling user application enters the system.

Name	sb_exit
Arguments	none
Return Value	none
Description	signals that the calling user application will leave the system.

Name	sb_connect_appl
Arguments	appl_name: **SbTName**
Return Value	appl_ref: **SbTApplication**
Description	establishes a logical connection with *appl_name*.

Name	sb_disconnect_appl
Arguments	appl_ref: **SbTApplication**
Return Value	none
Description	destroys the logical connection to *appl_ref*.

Name	sb_id
Arguments	name: **SbTName** parent_ref: **SbTStiParent**
Return Value	obj_ref: **SbTSti**
Description	obtains the reference of (a proxy of) the object identified by *name* and *parent_ref*.

Name	sb_delete_obj
Arguments	obj_ref: **SbTSti**
Return Value	none
Description	deletes the object identified by *obj_ref*.

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 successfully establish a logical connection, the remote application (identified by *appl_name*) must have entered the system prior to the operation **sb_connect_appl**. The operations **sb_id** and **sb_delete_obj** are examples of object manipulation operations. 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.

2.2 Decomposition of the System

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

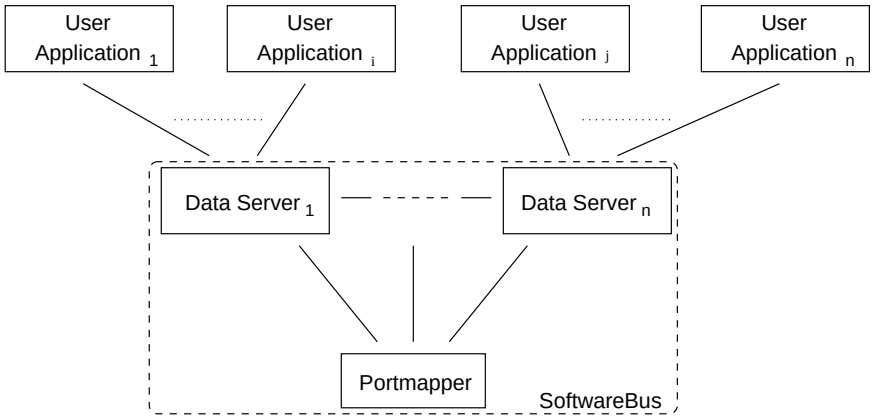


Figure 2: Decomposition of the **SoftwareBus**.

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. Within such a 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 that is kept on other servers is requested or needed. The number of data servers and their locations need not be 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 used in 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 four operations:

Name	pm_initialize
Arguments	appl_name: **SbTName**
Return Value	appl_ref: **SbTApplication**
Description	registers the application with the portmapper, so that other applications may find it and communicate with it.

Name	pm_exit
Arguments	appl_ref: **SbTApplication**
Return Value	none
Description	signals that the application may no longer be contacted.

Name	pm_connect_appl
Arguments	user_ref: **SbTApplication** server_name: **SbTName**
Return Value	server_ref: **SbTApplication**
Description	opens a connection to another application.

Name	pm_disconnect_appl
Arguments	user_ref: **SbTApplication** server_ref: **SbTApplication**
Return Value	none
Description	closes a connection to another application.

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**. We may think that all calls from user applications to the system are delivered by a data server. Upon receiving 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 to the correct user application. If we assume a one-to-one mapping between user applications and data servers, this additional parameter can be avoided (cf. Subsection 3.3).

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.

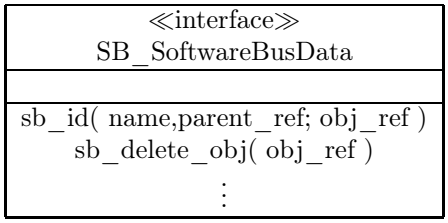
3 UML Specification

In this section, we present a specification of the **SoftwareBus** system using UML modeling techniques. We begin by providing static structural descriptions of the major system components such as interfaces, classes (or objects), and relationships among them. Static structures of the different classes of objects that may occur in the **SoftwareBus** system have already been described in the previous section. To provide a description of the system, we need to 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 components into UML diagrams.

3.1 External Interfaces

The external interface specifies operations provided by the **SoftwareBus** system to user applications. The operations of the external interface of the **SoftwareBus** system 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 provide their description using the UML graphical notation and describe their relationships to the **SoftwareBus** system.

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 that are necessary for object manipulation. Interpretations of signatures of the operations is as follows. In the list 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≫ SB_SoftwareBusConnections
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.

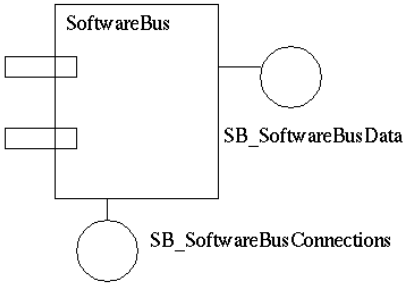


Figure 3: **SoftwareBus** interfaces.

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 user applications. The internal structure of the **SoftwareBus** system consists of the portmapper with a set of data servers. A description of the **SoftwareBus** system in terms of its relation to its classes is shown in a UML component diagram in Figure 4, with the two classes **Portmapper** and **DataServer**.

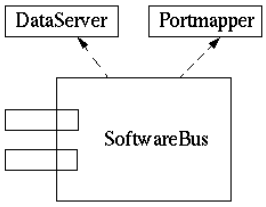


Figure 4: The **SoftwareBus** component and its classes.

Each of the classes has an interface, so we shall consider the two interfaces **SB_Portmapper** and **SB_DataServer**. The first one is the interface provided by the portmapper to data servers and the second one is the interface provided by data servers to other data servers. The interface **SB_Portmapper** is similar to SB_SoftwareBusConnections and is specified as follows:

≪interface≫ SB_Portmapper
pm_initialize(name;user_ref) pm_exit(user_ref) pm_connect_appl(user_ref,server_name; server_ref) pm_disconnect_appl(user_ref,server_ref)

The lists of parameters in these operations are extended by one argument with respect to the corresponding operations starting with **sb**. This additional argument is a reference to the process calling the **sb** operation and is passed to the *Portmapper* in order to identify the process. (See Section 3.3.)

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

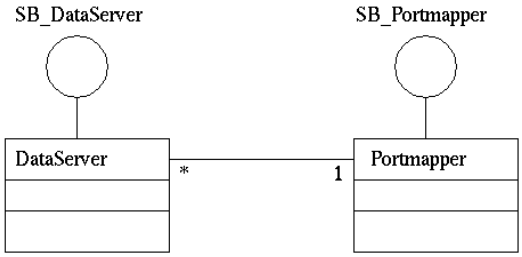


Figure 5: **SoftwareBus** class diagram.

Relation between Internal and External Components. Looking from inside out, we need an interface (referred to as SB_DataServerToUser) from data servers towards user applications. The operations of SB_DataServerToUser are the same as those of SB_SoftwareBusData.

3.3 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, as shown in Figure 6. If we assume such a mapping, parameters to method calls can be simplified, as the identity of the user application at the source of a call would be given by its corresponding data server. In this paper, we have adopted a more general solution, where we allow several applications to employ 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 may be implemented in one application program [2]. Therefore, in Figure 6, a user

application and its data server can often be thought of as one component, a Combined-Application. In such an implementation, the interface of Combined-

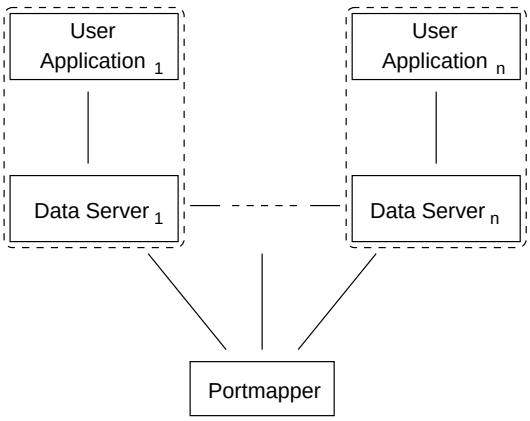


Figure 6: Structure of a **SoftwareBus** system with data servers integrated in applications programs.

Applications to other Combined-Applications is the same as that of data servers to other data servers, and the interface of the portmapper to Combined-Applications is the same as that of the portmapper to data servers.

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 [27]. In this section, we shall restrict the behavior of the UML interfaces of Section 3, and thereby obtain OUN interfaces that express the behavior we expect from the interfaces of the **SoftwareBus**.

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 $p_1, \ldots, p_n$ and returns with values $v_1, \ldots, v_m$. The call is represented by two distinct communication events: first, $o' \rightarrow o.m(p_1, \ldots, p_n)$ reflects the *initiation* of the method call and then, $o' \leftarrow o.m(p_1, \ldots, p_n; v_1, \ldots, v_m)$ 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. Remark that as OUN objects have an associated process, these objects are different from **SoftwareBus** objects. In the following section, we discuss the 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. In this section, we consider the behavioral constraints on the interfaces.

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:

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

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 "ε" 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 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:

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

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(caller,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 [14]. Although we have not specified any particular invariant predicate in the declaration of SB_SoftwareBusData, an assumption predicate P always implies the implicit invariant predicate P, for histories ending 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*).

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

In the specifications, **prp** is the prefix predicate. Let h **ptn** P denote that the trace h is specified by the pattern P. The predicate h **prp** P is true if there is an extension h' of h such that hh' **ptn** P, where hh' is the concatenation of the two traces h and h'. In the patterns, the notation $o_1{\leftrightarrow}o_2.m(\ldots)$ is shorthand for the initiation event $o_1{\rightarrow}o_2.m(\ldots)$ immediately succeeded by the completion $o_1{\leftarrow}o_2.m(\ldots)$. This corresponds to synchrony if the two events are perceived as immediately succeeding each other by both parties. For brevity we use the same notation in the set below to represent 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. The following predicate is used to determine whether an application a is registered in the **SoftwareBus** system:

$$
\begin{array}{l}
up(a, p, h) = \\
\quad h/\ \{a{\leftrightarrow}p.\mathrm{sb\_initialize}(), a{\leftrightarrow}p.\mathrm{sb\_exit}()\} \\
\qquad \mathbf{ptn}\ (a{\leftrightarrow}p.\mathrm{sb\_initialize}()\ a{\leftrightarrow}p.\mathrm{sb\_exit}())^*\ a{\leftrightarrow}p.\mathrm{sb\_initialize}().
\end{array}
$$

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:

$$
\begin{array}{l}
connection(a_1, a_2, p, h) = \\
\quad h/\{a_1{\leftrightarrow}p.\mathrm{sb\_c\_a}(\_;a_2), a_1{\leftrightarrow}p.\mathrm{sb\_d\_a}(a_2)\} \\
\qquad \mathbf{ptn}\ (a_1{\leftrightarrow}p.\mathrm{sb\_c\_a}(\_;a_2)\ a_1{\leftrightarrow}p.\mathrm{sb\_d\_a}(a_2))^*\ a_1{\leftrightarrow}p.\mathrm{sb\_c\_a}(\_;a_2).
\end{array}
$$

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 as a predicate on the history, checking that *up* holds before $\mathrm{sb\_c\_a}(a_2)$ is called and that *connection* holds before $\mathrm{sb\_d\_a}(a_2)$ is called.

$$
\begin{array}{l}
cn(p, \varepsilon) = \mathbf{true} \\
cn(p, h \vdash a_1{\leftarrow}p.\mathrm{sb\_c\_a}(\_;a_2)) = cn(p,h) \wedge up(a_1,p,h) \wedge up(a_2,p,h) \\
cn(p, h \vdash a_1{\leftarrow}p.\mathrm{sb\_d\_a}(a_2)) = cn(p,h) \wedge connection(a_1,a_2,p,h) \\
cn(p, h \vdash \mathrm{others}) = cn(p,h)
\end{array}
$$

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 $cn(\text{this}, h)$
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. The specification of interfaces for the portmapper and the data servers in OUN are discussed in the sequel.

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:

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

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.

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

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

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.

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

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 $cn'(\text{this}, h)$
end

The interface **SB_DataServer** is similar to SB_SoftwareBusData, so the description is omitted.

Relation between Internal and External Components. As explained in the previous section, the operations of the interface provided by data servers towards user applications (i.e. SB_DataServerToUser) are the same as those of the external interface SB_SoftwareBusData. However, the specifications of the predicates and invariants may be refined, as there are more details in the visible communication history of an object with the SB_DataServerToUser interface than that of the **SoftwareBus** system.

5 A Robust Portmapper

The **SoftwareBus** system, as we have specified it in the previous sections, is clearly prone to errors. In particular, the **SoftwareBus** system has a dynamic structure where user applications can initialize or exit the system at any time and objects can be (remotely) created, modified, and deleted. This may give rise to deadlock situations because 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 give rise to deadlock in the portmapper, to ensure that the communication framework is operative even in situations where some user applications have deadlocked. (In open distributed systems like the **SoftwareBus**, the robustness of user applications is outside our control.) Possible deadlock situations in the portmapper will be avoided by issuing *exceptions* in response to method calls from user applications when the regular behavior of the portmapper is out of place. In this section, we will consider some possible modifications of the specifications in order to make the portmapper more robust. We will follow the methodology outlined in [16, 17], 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 a given 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 normally expected 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.

As we have already mentioned, the portmapper might deadlock due to the dynamic environment it is set to control. In particular, a situation might arise where a user application tries to open a connection to another application after the latter has gone down. We will now consider the following error situations:

- $F1$: Uniqueness of application names. An application tries to register with a name that is already in use.
- $F2$: An application tries to open a communication channel to another application which is not available.
- $F3$: An application tries to close a communication channel to a server when the channel is already closed.

We refer to these error situations as fault classes $F1$, $F2$, and $F3$, respectively. In all three cases, we want the portmapper to return an exceptional completion event in response to a method call from a user application and continue as if the fault had not occurred. For this purpose, we modify the interface of the portmapper so that the deadlock situations are removed. How a fault will be handled eventually, depends on the user application that receives the error message. Therefore, the specification of the user application could (also) profitably consider exception events that are part of their alphabet, although fault handling in user applications will not be considered here.

Inside the **SoftwareBus**, the registration of applications is handled by the data servers. Define a function on the traces of the portmapper to calculate the set of names that are in use for applications of the **SoftwareBus** at a specific point in time:

$$
\begin{aligned}
&in\_use(\varepsilon) = \emptyset \\
&in\_use(h \vdash d{\leftarrow}p.\mathrm{pm\_initialize}(a; \_)) = in\_use(h) \cup \{a\} \\
&in\_use(h \vdash d{\leftarrow}p.\mathrm{pm\_exit}(a)) = in\_use(h) - \{a\} \\
&in\_use(h \vdash \mathrm{others}) = in\_use(h)
\end{aligned}
$$

A fault of class $F1$ occurs when an application name is already in use by another user application at the invocation of *pm_initialize*. If $x{\leftarrow}y.m(i_1, \ldots, i_n; \ldots)$ is a completion event in response to an invocation of a method m with input $i_1, \ldots, i_n$ and F is a fault class associated with that method, we represent by $x{\leftarrow}y.m_F(i_1, \ldots, i_n)$ an F-exception event raised in response to the same invocation of m. Define a predicate $reg(h)$ to express that exception events corresponding to faults of class $F1$ are issued correctly, as a predicate on the history of the portmapper:

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

Here, different output events are issued in response to calls to pm_initialize, depending on whether the name of the new user application is currently in use in the **SoftwareBus** system or not. (Remark that we can define similar predicates for references to applications, in order to strengthen the invariant of the portmapper interface.)

Next, we apply the same strategy in order to 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 when no communication link is established. Let cn'' be a modification of the cn' predicate, defined as follows:

$$
\begin{array}{l}
cn''(p, \varepsilon) = \textbf{true} \\
cn''(p, h \vdash a_1{\leftarrow}p.\text{pm\_c\_a}(\_, a_2; \_)) = up(a_2, h) \wedge cn''(p, h) \\
cn''(p, h \vdash a_1{\leftarrow}p.\text{pm\_c\_a}_{F2}(\_, a_2; \_)) = \neg up(a_2, h) \wedge cn''(p, h) \\
cn''(p, h \vdash a_1{\leftarrow}p.\text{pm\_d\_a}(\_, a_2)) = channel(a_1, a_2, p, h) \wedge cn''(p, h) \\
cn''(p, h \vdash a_1{\leftarrow}p.\text{pm\_d\_a}_{F3}(\_, a_2)) = \neg channel(a_1, a_2, p, h) \wedge cn''(p, h) \\
cn''(p, h \vdash \text{others}) = cn''(p, h)
\end{array}
$$

The predicates cn'' and reg regulate output from the portmapper, so together they will define the invariant of the **SB_RobustPortmapper** interface. The assumption predicate of the original **SB_Portmapper** interface will also need modification to include the exceptional completion events that are now allowed in the histories. We will assume that for the initialization of user applications and for opening communication channels, an application can attempt to repeat a call in the case of an exception. (For the other methods of the **SoftwareBus** system that are not considered in this paper, the behavior will often be more sophisticated.) The new assumption $correctComSeq''$ is defined as follows:

$$
\begin{array}{l}
correctComSeq''(a, p, h) = \\
\quad h \ \ \textbf{prp}\ (a{\leftrightarrow}p.\text{pm\_initialize}_{F1}(\_; \_)^*\ a{\leftrightarrow}p.\text{pm\_initialize}(\_; \_) \\
\qquad\quad [a{\leftrightarrow}p.\text{pm\_c\_a}_{F2}(\_, \_; \_)^*\ a{\leftrightarrow}p.\text{pm\_c\_a}(\_, \_; \_) \\
\qquad\qquad a{\leftrightarrow}p.\text{pm\_d\_a}(\_, \_)\ a{\leftrightarrow}p.\text{pm\_d\_a}_{F3}(\_, \_)^*]^* \\
\qquad\qquad\quad a{\leftrightarrow}p.\text{pm\_exit}())^*
\end{array}
$$

Given the modified predicates above, the robust portmapper interface is now specified:

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 $correctComSeq''(\text{caller}, \text{this}, h)$
  inv $cn''(\text{this}, h) \wedge reg(h)$
end

If we simply ignore failed method calls in the **SB_RobustPortmapper** interface above, i.e. exception events and the initiation events that correspond

to them, we have the intolerant interface **SB_Portmapper**. Consequently, a proof that the **SB_RobustPortmapper** interface is a fault-tolerant refinement of the **SB_Portmapper** interface is fairly straightforward by induction over traces [17].

We can now consider the data server in the presence of faults. Its behavior may vary depending on the functionality of the user application. To a certain degree, faults may even be kept within the **SoftwareBus**, not always penetrating to the user applications by the external interface. This gives us a kind of fault transparency for the **SoftwareBus**. The behavior of the data server when faults occur can be specified in the way outlined above, but possibly with more complicated fault handling. Even if the error situations cause deadlock in data servers or user applications, the portmapper can now handle these error situations, avoiding deadlock due to the fault classes discussed above. The functionality of the communication framework of the **SoftwareBus** system has become more robust, as errors are propagated to the user applications.

6 Discussion

In this paper, we have considered how a combination of graphical and formal specification notations can be used for the purpose of developing a dynamic communication framework. The framework we have specified is a simplified version of the **SoftwareBus** system developed at the OECD Halden Reactor Project, focusing on the basic infrastructure.

The basic purpose of communication is to share information and resources. In addition to transmitting existing data, an application may ask another application to provide other services. For instance, if an application needs the sum of an array, it would be better to ask the server to calculate and provide the sum instead of transmitting the whole array which may consist of many items. The contents of communication depends on the service requested and provided. Generally, the following aspects are important with respect to communication:

- *Functionality:* An application should have the possibility to create, modify, and read a piece of data (of any type), to create, modify, and read a class definition, to add attributes and methods to a class, and to invoke such methods in remote applications.

- *Data transparency:* Semantically equivalent data may be stored differently in different computers. An application should be able to interpret a piece of data independent of the computer that sends the data.

- *Uniformity:* Data and functionality must be accessible across applications in different computers in much the same way as they are available within a single process, so that an application programmer needs not be concerned about how to access data in different locations.

The simplified **SoftwareBus**, as specified in the previous sections, illustrates how to specify open distributed systems for our purposes of exchanging data between applications. However, it does not cover much of the necessary operations and functionality needed for actual applications. The complete OECD **SoftwareBus** provides a framework which offers this communication functionality, by means of specially designed methods for remote creation, modification,

and deletion of objects and classes. Data transparency can be achieved by using explicit class definitions for storage and interpretation of data. Uniformity can be achieved by implementing interface operations in such a way that they act according to whether the necessary data for the operations are in the local application or in a remote application. In the example of the **SoftwareBus** system, this can be handled by the local data server.

In this paper, we have started with graphical specifications in UML and developed a formal specification of a **SoftwareBus** system, concentrating on the central aspects of communication and openness present in the OECD **SoftwareBus**. In a rather straightforward manner, the UML interface specifications are extended into the textual specification language OUN by adding behavioral predicates to the specifications, in order to enable formal reasoning. The OUN interface language relies on explicit communication histories rather than state variables. This is attractive as it enables a black-box compositional style of reasoning about the observable behavior of objects and components. In the paper, we illustrate various ways of extracting information from the traces by means of predicates and patterns. The latter provide us with an intuitive, graphical specification style.

In OUN, further formal development of the specifications is possible. We can build more complete specifications of the objects from the interfaces we have specified in this paper. This can be done using the (multiple) inheritance mechanism provided for interfaces [16]. With our architectural lay-out of the **SoftwareBus** system, the data server objects are of particular interest, as they form the junction between the external and internal understanding of the system; i.e. the data servers are in communication with the user and the port-mapper as well as with other user applications connected to the **SoftwareBus**. Through inheritance, we can specify the intended interleaving of the behaviors captured in the different inherited interfaces. Also, OUN provides us with an incremental approach to system robustness [17], as illustrated in the case study.

In the general communication framework of the **SoftwareBus** system, facilities are provided for remote object creation and deletion as well as for code modification. Although the exact behavior of these facilities cannot be specified in detail in this general setting, they must be considered closely in the case of an adaptation of the framework to some specific need. OUN is a formalism designed especially for open, object-based, distributed systems, and provides facilities for reasoning about certain aspects of openness such as the transmission of object identities, class extension, and fault-tolerance within a formal framework. In the case study, we have captured and reasoned in OUN about the initialization of new user applications, communication channels between applications that open or close, and applications that exit the **SoftwareBus** system at any time during execution. Modifications of object behavior are specified in terms of predicates on communication traces. Hence, the formalism is well-adapted to capture the flexibility of the **SoftwareBus** system in a rigorous way which enables formal reasoning.

The approach to process development proposed in this paper 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. The use of UML constructs is important for the description of the initial software requirements which is normally a result of discussions between users and systems

analysts (or software engineers). By extension of UML interface specifications, we obtain specifications in OUN. The OUN specifications are used to capture dynamic aspects that are not easily expressible in UML or OCL [34].

Although OCL extends UML with object invariants and pre- and postconditions for methods, precise specification of output from objects is still difficult to express in OCL as OCL specifications are given in terms of object attributes and not in terms of communication events [20]. We cannot always assume that information about object attributes are available for reasoning about components in open distributed systems. In open distributed systems, it is natural to perceive the behavior of a component as locally determined by its interaction with the environment [1]. In OUN, specifications rely on the observable behavior of black-box components. For open distributed systems, it is often advocated that specifications should be object-oriented and concentrate on different viewpoints [11]. In OUN, viewpoints to objects are given in interfaces with behavioral constraints on communication. Furthermore, object-oriented specification languages such as UML and OCL do not have a formally defined semantics and proof system. They are as such less apt for formal reasoning, lacking precise notions of composition and refinement that are present in OUN. The OUN specification language is object-oriented including notions of inheritance and object identity, in contrast to process algebras like CSP [10], CCS [24], the π-calculus [25], LOTOS [5], and the Actor language [1] (which has identity, but not inheritance). In contrast to object-oriented formalisms such as Object-Z [31], Maude [23], and temporal logic based approaches [22], OUN focuses on aspects of the observable behavior of objects rather than on object attributes and state transitions. This is an attractive approach to the specification of open systems, where we should not assume that implementation details of components in the environment are available for reasoning.

The approach of this paper relies on the PVS proof environment [8, 28] as a tool for consistency checking and verification of specifications. For the mapping of UML class diagrams into the PVS specification language, the basic modeling constructs and conditions can be expressed in the PVS specification language in terms of functions and abstract data types [3, 4]. For further system development, the OUN interface specifications may be translated into PVS in order to formally verify properties such as the refinement of specifications. The basic OUN notions of alphabets, traces, specifications, composition, and refinement can be represented in PVS and the semantic trace sets for the OUN syntax can be translated into PVS in order to take advantage of the PVS theorem proving facilities [13, 14]. A framework for the consistency check was described in [32, 33] 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

In this paper, we have demonstrated an approach to the specification of open distributed systems, based on a combination of UML and OUN specification

techniques. The system consists of a central unit (the portmapper) and a set of applications, and provides a dynamic communication framework for the purpose of exchanging data and resources between applications. In this paper, a minimal piece of the system, capturing the basic infrastructure, is used to illustrate how to specify open distributed systems where applications connect and disconnect over time. In the specification, we have used graphical UML constructs for the specification of interfaces, classes, and relations between different components.

The class diagrams of UML can be expanded into OUN (interface) specifications by restricting the implicit history variables of communication calls. By way of first-order predicates for assumption and commitment (invariant) clauses, we capture the observable behavior of the components. Using OUN concepts, the specifications allow formal reasoning about specification properties such as refinement. The aspectwise specification formalism of behavioral interfaces used in the OUN language lets us capture certain forms of openness by textual analysis, as demonstrated in the case study. This case study has focused on interfaces and communication between objects implementing the interfaces.

For the development of open distributed systems, OUN is a well-suited complement to UML. UML is the de facto industry standard, with intuitive graphical notation, but lacks the formalization necessary for rigorous system development and the concepts needed to capture the dynamic aspects of systems such as the **SoftwareBus**. In OUN, emphasis is on reasoning control: reasoning is both compositional and incremental so software units can be written, formally analyzed, and modified independently, while we have control of the maintenance of earlier proven results. As demonstrated in the paper, the OUN formalism lets us capture the dynamic behavior of open distributed systems, that are 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. An outline of UML class diagrams semantics using PVS-SL. In *Proceedings of the 11th Nordic Workshop on Programming Theory (NWPT'99)*, Uppsala, Sweden, Oct. 1999.

[4] D. B. Aredo, I. Traore, and K. Stølen. Towards a formalization of UML class structure in PVS. Research Report 272, Department of Informatics, University of Oslo, Aug. 1999.

[5] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.

[6] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin-Cummings, Redwood City, CA, 1991.

[7] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, Massachusetts, USA, 1999.

[8] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, Apr. 1995. Available, with specification files, at `http://www.csl.sri.com/wift-tutorial.html`.

[9] 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.

[10] C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice Hall, Englewood Cliffs, NJ, 1985.

[11] ITU Recommendation X.901-904 ISO/IEC 10746. *Open Distributed Processing - Reference Model parts 1–4*. ISO/IEC, July 1995.

[12] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

[13] E. B. Johnsen and O. Owe. A proof environment for partial specifications in OUN. In *Proceedings of the Norwegian Informatics Conference*. Tapir, 2001. To appear.

[14] E. B. Johnsen and O. Owe. A PVS proof environment for OUN. Research Report 295, Department of informatics, University of Oslo, 2001. The PVS theories are available at `http://www.ifi.uio.no/~einarj/OunPvs.tar.bz2`.

[15] 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 Publishers, Mar. 2002. To appear.

[16] E. B. Johnsen, O. Owe, E. Munthe-Kaas, and J. Vain. Incremental design of dependable systems in OUN. Research Report 293, Department of informatics, University of Oslo, 2000.

[17] 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. To appear.

[18] E. B. Johnsen, W. Zhang, O. Owe, and D. Aredo. Specification of distributed systems with a combination of graphical and formal languages. In *Proceedings of the Asian Pacific Software Engineering Conference (APSEC 2001)*. IEEE Computer Society, Dec. 2001. To appear.

[19] 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.

[20] A. Kleppe and J. Warmer. Extending OCL to include actions. In A. Evans, S. Kent, and B. Selic, editors, *Proceedings of UML 2000 - The Unified Modeling Language. Advancing the Standard*, volume 1939 of *Lecture Notes in Computer Science*, pages 440–450, York, UK, Oct. 2000. Springer-Verlag.

[21] F. Kostiha. 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)*, pages 146–156, Stockholm, Sweden, May 2000.

[22] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[23] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[24] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[25] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, May 1999.

[26] Object Management Group. *OMG Unified Modeling Language Specification*, June 1999.

[27] O. Owe and I. Ryl. OUN: a formalism for open, object oriented, distributed systems. Research Report 270, Department of informatics, University of Oslo, Aug. 1999.

[28] 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.

[29] 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.

[30] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, USA, 1991.

[31] G. Smith. *The Object-Z Specification Language*. Advances in Formal Methods. Kluwer Academic Publishers, 2000.

[32] 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.

[33] I. Traore, D. B. Aredo, and K. Stølen. Tracking inconsistencies in an integrated platform. Research Report 274, Department of Informatics, University of Oslo, Aug. 1999.

[34] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Object Technology Series. Addison-Wesley, 1999.