Specification of Distributed Systems with a Combination of Graphical and Formal Languages *

Einar B. Johnsen
Department of Informatics
University of Oslo, Norway
einarj@ifi.uio.no

Wenhui Zhang
Institute for Energy Technology
N-1751 Halden, Norway
wenhuiz@hrp.no

Olaf Owe
Department of Informatics
University of Oslo, Norway
olaf@ifi.uio.no

Demissie B. Aredo
Institute for Energy Technology
N-1751 Halden, Norway
demissie@hrp.no

Abstract

Convenience in specification and possibility for formal analysis are, to some extent, exclusive aspects of system specification. This paper describes an approach that emphasizes both aspects, by combining UML with a language for observable behavior of interfaces, OUN. These 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.

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. It is also desirable 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 combine existing formal methods and tools into a platform for specification, design and refinement of open distributed systems. For this purpose, we integrate the UML (Unified Modeling Language) modeling constructs [3], the OUN (Oslo University Notation) specification language [8] and the PVS (Prototype Verification System) theorem prover [9].

UML is a comprehensive notation for creating visual models of systems and has become a standard for object-oriented software development. It provides notations needed to define a system with any particular architecture, but lacks a formal semantics. *OUN* is a formal object-oriented design language for the development of open distributed systems. The language is designed in a restricted way so that reasoning is manageable, particularly, reasoning control is based on static typing and proofs, and generation of verification conditions is based on static analysis of pieces of programs or specification texts. Here, we only consider interface specifications in OUN. OUN objects may have internal activity, run in parallel, and communicate asynchronously. They support interfaces that describe observable behavior by means of input/output driven assumption guarantee specifications, using patterns on finite communication traces. An object may support a dynamically changing number of interfaces.

The integration of the techniques lets us exploit the advantages of graphical notations with UML, the formal notation of OUN, and the theorem-proving capabilities of PVS for verification of correctness requirements. The development process in our approach consists of the following mostly machine-assisted steps: informal specification of user requirements; partial specifications in UML; extension of UML interface specifications into OUN specifications; translation of the partial specifications into the PVS language, so that verification and validation can be done with PVS tools; and finally code generation.

This approach is demonstrated by a specification of the **SoftwareBus** system, which is an object-oriented data exchange system developed at the OECD Halden Reactor Project [1]. A full version of this paper is available [6].

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

2 Functionality of the Software Bus

Our main motivation for constructing a distributed software bus arises from the need for surveillance and control of processes in power plants. Data collected from processes have to be processed and presented in different forms at different locations, so data sharing among different user-applications is a necessity. Any number of user-applications can connect to the system to carry out data processing tasks, such as creation and destruction of variables, assignment of values to variables, and accessing values of variables (Figure 1).

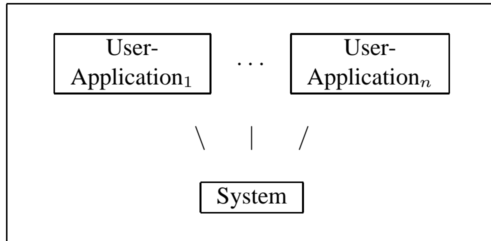

Figure 1. The Software Bus

The **SoftwareBus** system is object-oriented and open. Classes, functions, and variables are treated as **SoftwareBus** objects, i.e. manipulatable units in the system. Due to a hierarchical organization, objects are either referenced directly or identified by name and parent's identifier. Simplifying, we have three basic types: **SbTName** for names, **SbTSti** for identifiers, and **SbTApplication** for applications. There are subtypes of **SbTSti**, including **SbTStiParent**. The system interface towards user-applications provide the following operations:

- **sb_initialize**, with argument *name: SbTName*, no return value.
- **sb_exit**, no arguments or return values.
- **sb_connect_appl**, with argument *app_name: SbTName* and return value *app_ref: SbTApplication*.
- **sb_disconnect_appl**, with argument *app_ref: SbTApplication*, no return value.
- **sb_id**, with arguments *name: SbTName* and *parent_ref: SbTParent*, returns *obj_ref: SbTSti*
- **sb_delete_obj**, with argument *obj_ref: SbTSti*, no return value.

The operations **sb_initialize** and **sb_exit** are invoked by a user-application in order to enter and leave the system, whereas **sb_connect_appl** and **sb_disconnect_appl** concern logical connections between processes, and **sb_id** and **sb_delete_obj** are examples of object manipulation operations. For brevity, other operations are omitted here.

Decomposition of the System. It is preferable to keep data in, or near, the user-applications that possess the data, and provide a mechanism for decentralized data sharing. The system consists of a central unit, called a portmapper, and a set of data servers. The portmapper maintains information about data servers, while the servers store data to be shared among user-applications. Now, 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 portmapper if information concerning 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. For simplicity, a one-to-one mapping between user-applications and data servers is assumed.

Two interfaces are specified; an interface of the portmapper towards data servers and another of a data server to other data servers. The former interface includes the four operations *pm_initialize*, *pm_exit*, *pm_connect_appl*, and *pm_disconnect_appl*, which are internal equivalents to similar operations starting with **sb**. Upon receiving a call **sb_**m (for appropriate m) from a user-application, the data server forwards the call to the portmapper by calling **pm_**m. The operations of the interface of data servers are directly concerned with data manipulation, comprising methods such as *pm_id* and *pm_delete_obj*. Data servers issue calls to other data servers when necessary.

3 UML Specification

We specify the **SoftwareBus** system with UML modeling techniques. First, we provide static structural descriptions of major system components as described in the previous section, with classes, components, and the interfaces that they provide to each other. Then, static structure and dynamic behavior of the system is specified by putting these together in UML diagrams.

The external interface specifies operations that the **SoftwareBus** system provides as a service 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 connections between user-applications and the **SoftwareBus** system. Accordingly, we decompose the interface towards external user-applications into two sub-interfaces: **SB_SoftwareBusData** and

SB_SoftwareBusConnections. For the purpose of this paper, the interface **SB_SoftwareBusData** includes at least the operations **sb_id** and **sb_delete_obj**. Relationships between the software bus system and its external interfaces are given in Figure 2.

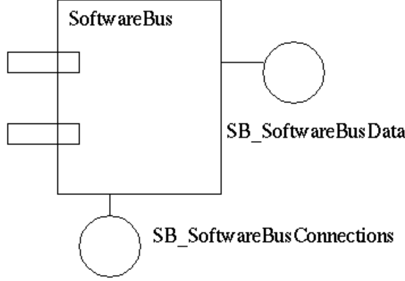

Figure 2. External Interfaces

"Internal" interfaces are provided by the components of the **SoftwareBus** system to each other. The portmapper offers **SB_Portmapper** to data servers and data servers offer **SB_DataServer** to other data servers. SB_DataServer resembles SB_SoftwareBusData, SB_Portmapper is specified as follows:

≪interface≫ SB_Portmapper
pm_initialize(name) pm_exit( ) pm_connect_appl(user_name; user_ref) pm_disconnect_appl(user_ref)

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. Types of the parameters are as specified in the previous section.

4 OUN Specification

Interface declarations in OUN are used to specify relevant aspects of the *observable behavior* of objects. The interfaces of an object are specifications concerning aspects of the behavior of that object, i.e. the possible communication histories of the object when a particular subset of its alphabet is taken into account. Thus, an object considered through an interface is a specific viewpoint to the services provided by the object. There may be several interfaces associated with an object, which give rise to different specifications of that object.

An interface identifies a set of communication events that reflect method calls relevant to a role of the object. Say that a method m, provided by an object o, 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 communication events: $o' \rightarrow o.m(p_1, \ldots, p_n)$ reflects the initiation of the method call and $o' \leftarrow o.m(p_1, \ldots, p_n; v_1, \ldots, v_m)$ reflects the completion of the call. The behavior of the aspect of the object that is modeled, is given by (first-order) predicates on finite sequences of such communication events. For each interface there are two predicates, an *assumption* and an *invariant*. The assumption states conditions on the environment of the object, so it is a predicate that should hold for sequences that end with input events to the object. 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. In this section, we consider an OUN specification of the **SB_Portmapper** interface.

We define predicates on communication traces to express behavior (abbreviating method names). Let p be an object which offers the SB_Portmapper interface, i.e. p is the portmapper. Let h be the history of p. We *assume* that data servers initialize, connect and disconnect to other servers and finally exit the system. This is expressed as a *prefix of a pattern*, where patterns are regular expressions:

$$
\begin{aligned}
&correctComSeq(d, p, h) = \\
&\quad h\ \textbf{prp}\ [d \leftrightarrow p.\mathrm{pm_initialize}(_, _) \\
&\qquad [d \leftrightarrow p.\mathrm{pm_c_a}(_, _; _)\ d \leftrightarrow p.\mathrm{pm_d_a}(_, _)]^* \\
&\qquad\ d \leftrightarrow p.\mathrm{pm_exit}(_)]^*
\end{aligned}
$$

Here, a $\leftrightarrow$ event denotes a $\rightarrow$ event immediately succeeded by a $\leftarrow$ event, ε is the empty trace, and $\vdash$ is right append. Using similar techniques, we define a predicate up to determine whether a server is initialized and $connection$ to determine whether two servers are connected. In the software bus system, applications connect and disconnect to each other. Two applications should not attempt to disconnect unless they already have an open connection. Let s_1 and s_2 be data servers. Using definition by cases, this is expressed by a predicate on h as follows:

$$
\begin{aligned}
&cn(p, \varepsilon) = \textbf{true} \\
&cn(p, h \vdash d \leftarrow p.\mathrm{pm_c_a}(s_1, _; s_2)) = \\
&\qquad cn(p, h) \wedge up(s_1, p, h) \wedge up(s_2, p, h) \\
&cn(p, h \vdash d \rightarrow p.\mathrm{pm_d_a}(s_1, s_2)) = \\
&\qquad cn(p, h) \wedge connection(s_1, s_2, p, h) \\
&cn(p, h \vdash \mathrm{others}) = cn(p, h)
\end{aligned}
$$

Let "this" object provide the SB_Portmapper interface and "caller" range over data servers. Using the predicates above, the interface SB_Portmapper can be specified in OUN 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
```

5 Discussion

We have demonstrated an approach based on a combination of UML and OUN specifications to the specification of a distributed system. The system consists of a central unit (the portmapper) and a set of applications, with the purpose of exchanging data. In this paper, a minimal piece of the system is used to illustrate how to specify open distributed systems, with dynamic communication patterns and remote object creation. In the specification, graphical UML constructs are used to specify interfaces, classes and relations between different components.

As demonstrated in the paper, UML class diagrams 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. In the case study, we have focused on interfaces and communication between objects that implement these interfaces. The aspect-wise 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. Further details are found in the full paper [6].

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 to describe the initial software requirements which are 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 advantage of using OUN is that OUN specifications express observable behavior of objects and captures dynamic behavior which is not easily expressible in UML and OCL [10]. The OUN specification language is object-oriented including notions of inheritance and object identity, and focuses on aspect-oriented specification of observable behavior, in contrast to process algebras like CSP [4], CCS [7], and LOTOS, as well as OCL, Z+, Object-Z, Maude, and temporal logic based approaches.

In the OUN specification language, 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. OUN uses a novel input/output driven assumption commitment specification style, by means of first-order predicates and graphical-style patterns on communication histories, thus emphasizing reasoning facility.

The approach of this paper relies on the PVS proof environment as a tool for consistency checking and verification of specifications. UML class diagrams can be mapped into the PVS specification language for consistency checking [2]. For further system development, OUN interface specifications may be translated into PVS in order to formally verify for instance refinement properties of the specifications [5]. Code generation facilities for OUN specifications are currently under development.

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] T. Akerbæk and M. Louka. The software bus, an object-oriented data exchange system. Technical Report HWR-446, OECD Halden Reactor Project, Inst. for Energy Technology, Norway, 1996. See also `http://www.ife.no/swbus`.

[2] D. B. Aredo, I. Traore, and K. Stølen. Towards a formalization of UML class structure in PVS. Technical Report 272, Dept. of Informatics, Univ. of Oslo, 1999.

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

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

[5] E. B. Johnsen and O. Owe. A proof environment for partial specifications in OUN. In *Norwegian Informatics Conference*. Tapir, 2001.

[6] E. B. Johnsen, W. Zhang, O. Owe, and D. B. Aredo. Combining graphical and formal specification: the software bus case study. Technical Report 297, Dept. of informatics, Univ. of Oslo, 2001.

[7] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[8] O. Owe and I. Ryl. On combining object orientation, openness and reliability. In *Norwegian Informatics Conference*. Tapir, 1999.

[9] S. Owre, N. Shankar, and J. M. Rushby. *The PVS Specification Language*. Computer Science Laboratory, SRI International, 1993.

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