Correctness of distributed systems with middleware.

Hanmei. Cui
University of Windsor

Correctness of Distributed Systems with Middleware

by

Hanmei Cui

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through the School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science
at the University of Windsor

Windsor, Ontario, Canada
April 2003

Your file Votre référence
ISBN: 0-612-82861-1
Our file Notre référence
ISBN: 0-612-82861-1

Abstract

Distributed systems are complex and expensive to develop. One of the most difficult issues comes from the fact that the processes in a distributed system may be working on different platforms and in different programming languages. Middleware technologies have been developed to handle the heterogeneity in the complex distributed communications and provide us with high-level primitives for inter-process communication, while hiding the details of its network management and services. By using a proper middleware, the software components can be made accessible across applications and machine boundaries. By nature, middlewares have some limitations such as event-reordering, limited capacity for message-withhold, etc. Such limitations make the distributed applications error-prone. Here we present a formal verification technique for the correctness of the design models of the component-based distributed applications where the inter-process communications are realized via *general* middleware layers.

We give a suitable extension of UML statechart diagrams. With this extension, the components and the communications among them in a middleware-based distributed application can be clearly expressed, and the software developers can actually view the middleware as a black box. We assume that the abstract behavior of the distributed application is given in such notations using any UML tool capable of saving the design in the standard XMI format. Two specially formatted deployment files are used to set up a concrete system from the design specification. The verification of its correctness is achieved by converting the design model, together with the deployment details, into a formal verification model in specification language PROMELA. The derived formal description is composed of the translation of the specification of the application itself and of the in-depth modeling of the behavior of the adopted middlewares, including their undesirable side effect. With the formal description, the related model checking tool SPIN can then be applied to verify the system and/or to simulate the system behavior as a way of debugging. In doing so, we are able to single out in the distributed applications, any design errors, especially for those related to or caused by the middleware layer, via the formal verification techniques.

Keywords:

Distributed Systems, Middlewares, Formal Specification and Verification, Model Checking, SPIN, UML, CORBA, RMI, JMS, MOM.

Acknowledgement

I would like to take this opportunity to thank my supervisor, Dr. Jessica Chen, for her invaluable guidance and advices, for her enthusiastic encouragement and her great patience to me. Without her help, the work presented here would not have been possible.

I would also like to thank my committee members, Dr. Arunita Jaekel, Dr Zhiguo Hu and Dr. Angela Sodan for spending their precious time to read this thesis and putting on their comments, suggestions on the thesis work.

My special thanks go to the secretaries of the School of Computer Science, Ms. Mary Mardegan and Ms. Margaret Garabon, for their consistent helps. Thanks also go to my classmates, Xiubin Cai, Yongdong Tan, for their help.

I would also like to thank my husband, Jun Lin, for his understanding, patience and support, and thank my daughter, Emily, for being such a good baby.

Table of Contents

Chapter 1. Introduction

In this chapter, we address the problems that have motivated our research work and outline our solution for such problems. In section 1.1, we explain the rationales for specifying the distributed systems at two different abstract levels. In section 1.2, we introduce middlewares and point out the pitfalls when modeling them. In section 1.3 we outline our research and introduce the structure of this thesis.

1.1 Distributed Systems and Middlewares

A distributed system is an application whose functionality is distributed among multiple processes possibly residing on physically separated machines across the network. Such applications exploit the networked multi-processing power and/or offer advantage like resource sharing, fault tolerance, etc.

Processes in a distributed application need to coordinate with each other through the Inter-Process Communication (IPC). IPC is often built up on top of the networks, and this raises many issues such as security, reliability, efficiency, and heterogeneity. To minimize the security vulnerabilities, guardian services such as identification, authentication and authorization must be provided so that the system can be well protected from abusive users, hackers, virus, etc. To boost reliability, proper error detection and error recovery routines should be in place in case a message is distorted, lost or duplicated due to network noise. To name, describe and manage the information about the distributed resources, directory services act as an important component in a distributed architecture in terms of efficiency.

Network services such as these are not application-specific. They can be provided by a layer of third-party software between the network and the applications. This software layer is called *middleware*.

In addition, middleware handles the heterogeneity in the complex distributed environment and provide the programmers with high-level, intuitive IPC primitives, hiding the details of its services beneath the Abstract Programming Interfaces (APIs). Distributed events, remote procedure calls, transparent message channels are all examples of such primitives. A great variety of middlewares exists in our market today. They differ a lot in cost, size and range of services they provide. If chosen properly and used correctly, middlewares can greatly reduce the development cost and enhance the reliability, scalability, maintainability and interoperability of the applications.

Most middlewares developed for general distributed applications belong to two middleware families: message oriented middleware (MOM) and distributed object-based middleware (DOS).

The basic form of IPC is the point-to-point (PTP) message passing. MOMs simplify yet do not mask the IPC message passing. With MOMs, the sender and the receiver of a message may refer to each other through some universal identifiers instead of the physical locations. In addition to the PTP style message passing, more sophisticated MOMs may provide a publisher/subscriber system (also called distributed event system) built on top of the PTP message passing to enable message multicasting. In such systems, the senders and the receivers are de-coupled. The IPCs in MOM are typically asynchronous.

Most MOMs are used for workflow, process-control applications, and wide-area network applications with slower, less reliable connections. Message queuing system is a typical kind of persistent PTP communication system. Java Messaging Service (JMS) is a general MOM that describes both the PTP and the publisher/subscriber message passing mechanisms [19].

A major problem with MOM is that the messages or the distributed events are usually not expressive enough. The content of a message, if examined alone, does not show much about the meaning of the message: One cannot be sure whether it is a service request, or a database query, or a plain text string as data. The sender and receiver must first agree upon the format and even the ordering of the messages so that the receiver can interpret the message correctly. Therefore, the logic of the application can be clouded by the interpretation of the IPC messages.

DOS middlewares, on the other hand, provide a "distributed object-based system" on top of the PTP message passing. An *object* in a distributed object system is an encapsulated entity that can be uniquely identified. It provides one or more services (operations) that can be requested (called) by a client residing in a different process. A client is an entity capable of requesting the service. A client is often by itself a distributed object. Instead of using the plain messages, the IPC takes the form of *service requests*. The information associated with a request consists of an operation, a target object, zero or more (actual) parameters, and an optional request context. A request is also called a remote procedure call (RPC) or remote method invocation (RMI). The client and the server object do not have to know the physical locations of each other: The DOS middleware is responsible for interpreting the requests, finding the server object for the request, preparing the server object to receive the request, passing the request data to the server, and for passing the result back to the client. The DOS middleware services are transparent to the applications: the syntax and semantics of a request is similar to the local procedure calls.

As we can see, middlewares greatly simplified the IPC. Consequently, however, they also tend to mask the problems associated with the network communication. For example, although middlewares usually have guards against the message duplication, message alternation, message lost, and message reordering cannot be filtered entirely out by middlewares. Such phenomena may affect the correctness of the business logic of the applications. In addition, a middleware is of limited capacity, and the IPC service it provides deteriorates when the limit is reached. When the buffer of a communication channel is full, for example, new messages coming to the channel will be either dropped or blocked. When a server object in a distributed object system runs out of thread resources, the new requests to the object will be blocked, and this may raise the deadlock and livelock problems.

Such faults and errors are hard to identify, hard to catch, and are very often overlooked. This is because (i) The middleware APIs distances the programmers from the reality of the network communications; and (ii) The faults and errors only appear occasionally.

1.2 Model Checking and UML Specification

The present work is dedicated to the investigation and development of a model checking-based verification technique for the correctness of the design models of the distributed applications where the IPCs are realized via the underlying middlewares. Model checking has been recognized as a suitable and effective technique developed for the systematic examination of the design specifications [4, 7, 8, 9, 14]. In a distributed system, processes are normally executed concurrently, thus existing formal verification techniques on concurrent systems can be adopted into this setting. However, with the inherent complexity of the network communication, distributed systems have imposed new challenges to the study and the investigation of formal verification techniques. With the present work, we are able to diagnose the distributed applications at the design stage and check whether a given design specification satisfies the desired properties. In particular, whether it is free from phenomena such as resource and communication deadlock, live-lock, starvation, race conditions, etc.

Formal verification techniques rely on formal specifications of the design documents. As we consider the verification of the correctness of the design models rather than their underlying middlewares, this implies that the design models need to be formally specified. The fact is that, no matter which formal specification language we use, it turns out to be hard to grasp by an ordinary software developer who may not have the required expertise in it. One of the possible solutions here is to define an automated translation from some easy-to-use informal design models into the formal models.

UML is an industrial standard for designing object-oriented applications. The semi-formal UML notations, consisting of both graphical representations and natural language descriptions, can be easily understood and effectively communicated with by software developers. Here we assume that the design specifications are given in UML style. Since the UML notations stretch over almost all aspects of software artifacts, translating the entire domain of UML notations is not feasible. Similar to other UML translation attempts [12 13 18], we choose to translate an essential subset of formalized UML class diagrams and statechart diagrams, as they typically provide enough information about the dynamic behavior of the model. As there is no default notation in UML for expressing distributed communications, we define an XML-based deployment file to specify the underlying middleware. Correspondingly, we enrich the UML class diagrams and UML statechart diagrams with a set of IPC interfaces that can be plugged into any UML specifications to express the IPCs via MOM or DOS middlewares. In order to perform the verification, we also define a set of special notations for expressing the correctness properties of the model.

An automated translation is provided to transform such a given design specification into an equivalent one in formal specification language PROMELA [8]. One of the most important features of our translator is that there is no need to directly describe the communication services provided by the MOM layer in the UML design model. The design model only shows the MOM APIs, and the translator will incorporate the communication services automatically to generate a verification model with the MOM services integrated. In fact, we provide PROMELA models of the message passing in both PTP and Publisher/Subscriber systems, with the main features including priority, persistency, resource limitation, message ordering and re-ordering. These models are integrated into the translated PROMELA models for the MOM-based applications.

The integrated formal specification forms the basis for the formal reasoning of the correctness of the applications. The generated PROMELA code includes the correctness requirements such as assertion statements and the SPIN [8,9] tool can be used on it to verify the satisfiability of those requirements. In doing so, we can catch design errors which include those related to or caused by the MOM layer.

In the following, we first give a brief introduction of the abstract the behavior of MOM and DOS middlewares. Secondly, we present the extension of UML for modeling a distributed system with such middlewares. Thirdly, we explain how to translate into PROMELA the part of the distributed system that is not related to IPC. Then we introduce the translation of middleware-related part of the UML model into PROMELA. At the end of the thesis we compare our work with others and indicate the future work that need to be done.

Chapter 2 Background: MOM and DOS Middlewares

As we explained in Chapter 1, the essential role of middlewares is to manage the complexity and the heterogeneity of distributed infrastructures and thereby provides a simpler programming environment for distributed-application developers. A middleware encapsulates the heterogeneity of the distributed computing environment by its universal API. Middleware API also provide higher-level building block for interprocess communication. A proper middleware can greatly reduce the development cost.

For general distributed systems, most middlewares belong to one of the two middleware families: message oriented middlewares or distributed object system middlewares. In this chapter, we give a brief introduction of the communication services provided by MOM and DOS middlewares, laying the ground for the modeling of their IPC interfaces in Chapter 3 and the modeling of their behaviors in Chapter 5 and Chapter 6.

2.1 MOM

Since distributed event systems are built on top of PTP message passing, PTP and distributed event systems share some common features. Here we explain the typical architecture of MOMs.

MOMs are usually configured as a set of channels, which can hold any type of messages, as system resource. MOM messages are often assigned with priorities. Messages with higher priority may arrive ahead of a lower priority message, even if the latter is sent earlier. The delivery of a message can be specified either as persistent (guaranteed delivery) or as transient (the middleware tries its best effort to deliver but does not guarantee the arrival of the message). In some MOMs, a transient message may arrive ahead of a persistent message, even if they are of the same priority and the latter was sent earlier. That is because a persistent message will go through additional buffering stage.

Another important concept in MOMs is session. A session is associated with a channel. It represents the partial message ordering service provided by MOMs: Messages sent through the same session are guaranteed to be retrieved in the same order as they were sent, except for the message re-ordering caused by message priority or persistency.

Senders and publishers are created based on sessions. A message sent from a sender session can only be retrieved once, but it can be browsed, that is, read without being removed. A message sent from a publisher session is multicasted to a number of subscribers, each subscriber gets its own copy of the message. Subscribers should be able to consume events at their own pace: a slow subscriber should not block a faster one.

When the resource limit is reached (e.g. the channel buffer is full), MOMs usually drop or block the new messages.

From the user's viewpoint, the channels provided by MOM can be considered as local buffers. In reality, however, such channels are built on top of the network communications, and thus phenomena such as message re-ordering, message-lost, etc. should not be overlooked.

In our modeling of the MOMs, we are not interested in services irrelevant to verifying the correctness of the applications built on top of them. Thus, for simplicity, we omit the services such as the encryption and the decryption of a message, identification and authentication, etc. during the modeling of MOM.

We model sessions, message priorities, transient and persistent messages, message browsing/receiving, message multicasting, and resource limit. Limited by the expression power of PROMELA, the channels are typed, and the message reordering among transient and persistent messages is not modeled. To simplify the model, we do not consider exceptions. We only consider the *ordinary* distributed event in our model, that is, a message without time or sequence related structure. Complex distributed events, which are based on particular message sequence, for example, event A = *received an Event B followed by an Event C without Event D in between,* are discussed mainly in theory and few middlewares actually support it.

2.2 DOS Middlewares

By using a DOS middleware, from the user's viewpoint, objects are made available by the middleware beyond process boundaries. A process can *plug* an object residing in another process, and probably on another machine, into itself as if the object belongs to it locally.

The DOS middlewares are standardized by OMG. The standard is called CORBA (The Common Request Broker Architecture) [16]. Most of the distributed object middlewares are CORBA-compatible. Those not compatible with CORBA, such as Java RMI or Microsoft DCOM, are similar to CORBA on the architecture level. In the present thesis, we only address CORBA-compatible middlewares in distribute object middleware category.

A CORBA distributed object system consists of one or more object request brokers (ORBs), a set of object implementations (objects that are available across the machine boundary) managed by some portable object adapters (POAs) and the clients.

An object implementation — or *implementatio*n for short — is a definition that provides the information needed to create an object and to allow the object to participate in providing an appropriate set of services to the clients. The ORB is responsible for all of the mechanisms required to:

- Find the object implementation for the request
- Prepare the object implementation to receive the request
- To manage the control transfer and data transfer to the object implementation and back to the client.

An ORB may be a single-threaded or multi-threaded. A single thread ORB will not be able to process another request before current request is transferred to the corresponding object adapter and, if the request is a synchronous call, the return value has been transferred back to the client. A single-threaded ORB is easier to implement but will likely create performance bottleneck and deadlock. Multiple threaded ORB is much more flexible.

An object adapter offers the primary way that an object implementation accesses the services provided by the ORB. The wide range of object granularities, lifetimes, policies, implementation styles, and other properties make it difficult for the ORB Core to provide a single interface that is convenient and efficient for all objects. Through *object adapters*, it is possible for the ORB to target the particular groups of object implementations that have similar requirements with the interfaces tailored to them.

Services provided by the ORB through an object adapter often include:

- generation and interpretation of object references.
- method invocation.
- security of interactions.
- object and implementation activation and deactivation
- mapping object references to implementations
- registration of implementations.

CORBA provides the specification for Portable Object Adapters (POAs), which standardized the interfaces exposed to the object implementations by the object adapters and other object adapter services. Since CORBA no longer supports Basic Object Adapters, in the rest of the thesis, we only consider portable object adapters.

We are interested in those services essential to verifying the correctness of the applications built on top of them. We simplify the generation and interpretation of object references in our model, and we will not discuss the security, registration, activation and deactivation of object implementations. What we are interested includes:

- binding: finding an object implementation for a request.
- method invocation:

    - Thread Resource
    - Synchronization

Binding means establishing the connection between proper object implementation and the client. It is the job of the ORB. The common criteria for binding includes:

- object id: The client may request the service of a particular object by its registered id, or by a name which can be translated by the naming service to the unique id.

- object type: The client may request the service of an object of particular type. Any object of that type will be acceptable to the client, while we can specify some additional restrictions such as the object adapter or the server process the object should be in.

Object adapters manage the object implementations. The most important job of object adapters is to perform method invocation. Each object adapter is assigned with some threads. Each method invocation is actually handled by a service thread, which takes input parameters, executes the remote procedure, and returns the result to the caller. If no thread resource is available at the time of invocation, the method invocation will be blocked until thread resource is available again.

The threads are assigned to an object adapter according to its thread policy. The most commonly used thread policies include:

1. *main_thread*. All object adapters with *main_thread* thread policy within an ORB will share the same thread, which is provided by the ORB.
2. *single_thread* (also called *thread_per_POA* in portable object adapters). The object adapter uses one thread to handle all method invocations to the objects it manages.
3. *thread_pool*. The object adapter uses a set of threads to handle method invocations. A thread returns to the thread pool after it served a method invocation.
4. *thread_per_object*. Each remote object in the object adapter will be assigned its own thread to handle the method invocations to the object.
5. *thread_per_client*: Each client of a remote object is assigned a thread.
6. *thread_per_request*: Each method invocation request is assigned with a thread. While this thread policy guarantees no blocking will occur within thread resource limit, it also leaves the server vulnerable to abusive users, since the server will crash when there is no more thread resource available in the system.

The client may invoke a method by different synchronization style. Typically, the communication in distributed object systems is *synchronous*: the client will wait until the remote method call is finished before continuing its execution. If the method invocation does not include return values, the method may be called *asynchronously*: the client will continue its execution as soon as the request is received on the other end without waiting for its completion. If the method invocation involves lengthy computation, *deferred synchronous call* is useful. In this invocation mode, the client continues execution after issuing the request, and pulls the result of the remote method call sometime later.

Our research models the object binding, the single thread and the multiple thread ORB, the invocation of the methods, the remote method service performed by threads under different thread models and different method invocation synchronization modes. To simplify the model, we only consider single thread ORB and multiple thread ORB with unlimited thread resource (never block). We assume that the communication between ORBs is instant, blocking-free and error-free.

Chapter 3 A Verifiable UML Design Model for Distributed Systems

3.1 Motivation

The goal of this chapter is to define a concrete and unambiguous UML model for distributed systems with middlewares. Such models will be translated into formal language PROMELA with a translator, which will be introduced in the following chapters.

As we pointed out in Chapter 1, UML is an industrial standard for designing object-oriented applications. The semi-formal UML notations, consisting of both graphical representations and natural language descriptions, can be easily understood and effectively communicated with by software developers. On the other hand, a UML specification is often neither executable nor verifiable due to the incompleteness and the ambiguity of such notations. This leaves the correctness of the specification in question, and has driven many researchers to turn to formal description techniques. A formal specification is by nature precise and unambiguous, and thus provides us with the basis to perform the verification against its correctness. However, formal specification languages turn out to be hard to grasp for an ordinary software developer who may not have the required expertise in mathematical and logical concepts.

One of the possible solutions here is to define a sound automated translation from the semi-formal UML models to formal models [12 13 15 18 21]. With such an approach, a formal specification model could be created directly from semi-formal graphical UML notations. If the translation process is automated, the consistency between the formal model and the UML model can be guaranteed, and the expertise needed to develop the system shifts from the formal specification techniques to UML. In particular, such an approach is desirable for distributed systems because:

(i) An object-oriented architecture is suitable for the distributed system design.
(ii) The specifications for distributed systems might be too complex to be examined by means other than formal verification.

A pre-condition for the aforementioned approach is that the designers must develop concrete and unambiguous UML diagrams. As the UML notations stretch over almost all aspects of software artifacts, the translation of the entire domain of UML notations may not be feasible. Thus, we consider translations based on a subset of UML diagrams to reduce the complexity. The most common choices for the subset are UML class diagrams and UML statechart diagrams, because these represent the structure and the behavior of the model. Other diagrams, such as UML deployment diagram, UML collaboration diagrams or UML sequence diagrams are sometimes used in conjunction with UML class diagrams and UML statechart diagrams either to set up a concrete system model, or to define correctness criteria. In the present work, we consider only UML class diagrams and statechart diagrams. We will mention later on, some other restrictions that we imposed on these diagrams in order to facilitate the automated translation, and to keep the formal model manageable.

3.2 Modeling Distributed Systems

We model a distributed system as an XML-style deployment file, a set of UML class diagrams (with restriction) and a set of UML statechart diagrams (with restriction).

Unlike some other attempts to model distributed systems with UML [5 8 10 11], our approach focuses on modeling the *behavior* of distributed systems in order to verify rather than to improve the formalness of the specification. We will emphasize on the interprocess communications and explicitly define a set of practical and formalized notations in our model for common IPC primitives provided by middlewares, namely the CORBA-style remote object systems and the MOM channels. The former includes ORBs, (portable) object adapters, remote objects and the client interfaces. The latter includes session-based point-to-point message channels and publish/subscribe-based distributed event channels.

In our design model, a distributed system is a set of processes that use IPC primitives provided by middlewares to coordinate with each other. We are interested in component-based processes, each containing a set of local components. Each local component is accessible directly only within the process boundary, and only through its interface. The communication among processes is carried out by either the DOS middlewares or the MOMs.

The deployment file configures the processes. Each process is specified as a set of local components. If the DOS middleware is used, a process may also host portable object adapters. The thread models of ORBs and POAs and the remote objects each POA manages are all specified in the deployment file. If MOM is used, the MOM channels are defined in the deployment file as *system resource*.

The behavior of local components and remote object components is specified in UML diagrams. The interprocess communication is carried out by calling functions on client interfaces of DOS middleware or MOM in such components. If the system uses DOS middleware, the remote objects are accessible application-wide through client interfaces in form of special *stub* and *response* objects (see Section 3.2.2). If the system uses MOM, the distributed event channels are accessible application-wide through client interfaces in form of special *publisher* and *Subscriber instance*s (see Section 3.2.2). Session-based point-to-point message channels are accessible application-wide through client interfaces in form of special *sender* and *Receiver instance*s (see Section 3.2.2).

In the following sections, we introduce the deployment file, followed by the special client interfaces we defined. Then, we explain the restricted syntax of UML class diagrams and UML statechart diagrams in our model, which defines general behavior of local components and remote objects.

3.2.1 The Deployment File

The role of the deployment file in our model is similar to that of a UML deployment diagram. It contains the information to construct a concrete system from the abstract model (classes) defined with UML class diagrams and UML statechart diagrams. We decided to define deployment files instead of customizing existing UML deployment diagram mainly because the latter does not fully support the design of distributed systems.

3.2.1.1 The Semantics of the Deployment File

The deployment file constructs the system in three steps.

First, we statically create a set of local components and a set of remote object components from the component interfaces and the IDL interfaces defined in the UML class diagrams. Each component is assigned a unique identifier.

Second, we set up the middlewares. We are interested in specifying the aspects related to the resource management of the middlewares, which affects the correctness of the business logic. For this reason, we include in the deployment file information such as thread policy, channel size, etc. and omit information such as security policy, etc.

- If DOS middleware is used, in the deployment file we specify a set of POAs and a set of ORBs. Each POA or ORB is given a unique identifier, and a thread policy. Each POA contains a set of remote object components. A POA resides in a process. The remote object component it manages can access the local components in that process through component interface. A POA also belongs to a particular ORB and accepts the service request from that ORB.

- If MOM is used, a set of distributed message channels will be defined in the deployment file. A distributed message channel is given a unique name and specified as either a publish/subscribe channel (a distributed event channel), or a session-based point-to-point message channel. In our model, channels are typed: Each channel can only carry the elements of a certain datatype. For simplicity, the buffer size is unique for all channels. The file also specifies the overflow policy, which applies to all point-to-point message channels. New messages coming to a full channel will either be dropped or be blocked according to the overflow policy.

Finally, from the deployment file, we can construct processes from the local components and POAs created above. Each process is specified as a set of local components and/or a set of POA references. Unless the process does not own any local components, the execution of a process starts with a method named *main*. A process can only have one *main* method and it must reside in a local component.

The physical deployment of processes is not shown in our model, because when a middleware is used, the physical deployment of the processes can be changed without influencing the functionality of the entire system.

3.2.1.2 The Syntax of the Deployment File

The deployment file in our model is a valid XML document [22]. The element types are defined in an external DTD file named deploy.DTD:

```
<!ELEMENT Deployment (Component_Set, Middleware, Processes)>
<!ELEMENT Component_Set (Component_Group)+>
<!ELEMENT Component_Group EMPTY>
<!ATTLIST Component_Group
              type       (local|remote)   #REQUIRED
              interface NMTOKEN       #REQUIRED
              id_range  PCDATA        #REQUIRED>

<!ELEMENT Middleware  (CORBA?, MOM?)>
<!ELEMENT CORBA (ORB)+>
<!ATTLIST ORB id ID #REQUIRED
                    thread_policy (multi_thread|single_thread) "multi_thread">
<!ELEMENT ORB (POA)+>
<!ELEMENT POA (Component)+>
<!ATTLIST  POA
              id ID #REQUIRED
              thread_policy  (main_thread | thread_per_POA |thread_per_client |
              thread_per_object | thread_pool_2 |thread_pool_3 |thread_pool_4 |thread_pool_5|
              thread_pool_6 | thread_pool_7 | thread_pool_8 | thread_pool_9 ) "thread_per_POA">

<!ELEMENT MOM (psChannel)?, (ptpChannel)? >
<!ELEMENT psChannel EMPTY>
<!ATTLIST psChannel
              name  NMTOKEN #REQUIRED
              buffer_size (1|2|3|4|5|6|7|8|9|10) "3"
              message_datatype (#PCDATA|int|short|byte|bool) #IMPLIED
              max_subscriber (1|2|3|4|5) "2" >

<!ELEMENT ptpChannel EMPTY>
<!ATTLIST ptpChannel
              name  NMTOKEN #REQUIRED
              buffer_size (1|2|3|4|5|6|7|8|9|10) "3"
              message_datatype (#PCDATA|int|short|byte|bool) #REQUIRED>

<!ELEMENT Process_Deployment(Process)+>
<!ELEMENT Process  ((Main_Class, Local_Components?)?, POA_IDs?), Environment?>
<!ATTLIST Process id ID #REQUIRED>
<!ELEMENT Main_Class EMPTY)
<!ATTLIST Main_Class class_name NMTOKEN #REQUIRED>
<!ELEMENT Local_Components (Component)+>
<!ELEMENT POA_IDs IDREFS>

<!ELEMENT Component (Initialization)+>
<!ATTLIST   Component id ID #REQUIRED>
<!ELEMENT Initialization (Attribute)+>
```

```
<!ELEMENT Attribute EMPTY>
<!ATTLIST   Attribute
            attName NMTOKEN #REQUIRED
            value    PCDATA #REQUIRED>
<!ELEMENT Environment (Signal)+>
<!ELEMENT Signal EMPTY>
<!ATTLIST Signal    type NMTOKEN #REQUIRED
                    max_subscriber    (1|2|3|4|5|6|7|8|9|10) "1" >
```

Figure 3.1 The deploy.dtd file

The file formally defines the syntax of the element types. In the rest of this section, we will explain the syntax and semantics of each element type in more detail.

3.2.1.2.1 The Header Element

The header of the deployment file is fixed:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE Deployment system "c:\mixup\deployment.dtd">
```

Figure 3.2 Deployment File Header

It shows that the dtd file is compatible with xml version 1.0 [22]. The path for the file *deployment.dtd* may need to be changed if c:\mixup is not the directory containing the file.

3.2.1.2.2 The Root Element

The root element must be a Deployment element. A Deployment element contains three child elements, namely a Component_Set element, a Middleware element and a Process_Deployment element. The Component_Set element is needed to create a set of component instances, including both local component and remote components. The middleware element is used to set up the middleware and the related IPC resources and the Process_Deployment element is used to create a set of processes from the components defined in the Component_Set element.

3.2.1.2.3 The Component_Set Element

The Component_Set Element is used to create component instances. Component instances are grouped by their interfaces. Accordingly, the Component_Set element contains one or more Component_Group elements. A Component_Group element has three attributes, namely *interface, type* and *id_range.*

Attribute *interface* is the name of an interface class defined in UML class diagrams. Since we allow only one class to realize a component interface in UML class diagrams

(See 3.3), the attribute *interface* uniquely identifies the behavior of the component.

The attribute *type* can take either value *local* or value *remote*. The value of the attribute *type* is related to that of the attribute *interface:* If the value of the attribute *interface* is a class name of a remote interface, then the value of *type* must be *remote*. This identifies the components defined in the Component_Group element as remote object components, which will be deployed to POAs and be accessible application-wide. On the other hand, if the value of the attribute *interface* is a class name of a local interface, the value of *type* must be *local*. This identifies the components defined in the Component_Group element as local components, which will be deployed to a process and will not be available across the process boundary.

The attribute *id_range* must be in the following format:

m_n

where m, n are positive integers with i<=j. It declares an integer range [m, n] for the identifiers of the components with interface type *interface*. We require that the id_ranges from different Components elements do not overlap with each other. Each reserved component id stands for a unique component instance. The initialization of the component, if necessary, is carried out later when deploying the component.

3.2.1.2.4 The Middleware Element

The Middleware element declares the interprocess communication facility. Since we support CORBA remote object invocation and message passing through MOM channels, the Middleware element may include a CORBA element and/or a MOM element, depending on the chosen middleware.

3.2.1.2.4.1 The CORBA Element

The CORBA element is used to set up CORBA ORBs and POAs. It contains a set of ORB elements. Each ORB element contains a set of POA element, an attribute named *thread_policy* and an *id*. The ORB ids must be continuous positive integers starting with "1". The ORB thread policy can be either *multi_thread* or *single_thread*. The former is the default value. The value *single_thread* identifies the ORB as a single_threaded ORB, which means all requests are processed serially by the ORB. The value *multi_thread* identifies the ORB as a multi_threaded ORB, which means that the requests can be processed by ORB concurrently.

Each POA element represents a Portable Object Adapter. A POA element contains two attributes, namely *id* and *thread_policy*. The id attribute uniquely identifies the POA. Its value must be in the following format:

p_I

where *p_* is the prefix, and *I* is any positive integer. No two POA can have the same *id* value.

The attribute thread_policy identifies the thread policy of POA. The value of attribute thread_policy can be:

POA_thread_policy value	**Description**
main_thread	All POA with POA_thread_policy valued *main_thread* shares a single working thread.
thread_per_POA	The POA uses a single working thread to handle all requests to the remote objects it owns.
thread_per_client	Each client to a remote object (a Stub instance) is assigned a working thread. Multiple service requests from the same client (from the same Stub instance) are queued.
thread_per_object	Each remote object the POA owns is assigned a working thread. Multiple service requests sent to the same remote object are queued.
thread_pool_n	The POA uses a working thread pool of size n to handle all service requests it received. 1<n<10. We restrict the pool size to 10 to reduce the complexity of the resulting model.

Figure 3.3 POA Thread Policy

The default value of POA thread_policy is *thread_per_POA*. Note that the last three values of the thread policy are not defined in CORBA specification: In CORBA specification, only single_threaded POA thread policies are standardized. These include main_thread and the thread_per_POA (also called single_thread) policies. The multi-threaded POA thread policy is left to the vendors. Here we have chosen to model three most commonly supported multi-threaded POA policies: thread_per_client, thread_per_object and thread pool.

A POA element contains one or more Component elements. Each Component element indicates to add the remote object component to the POA. A Component element contains an attribute named "id" and zero or more Initialization element. The attribute *id* is a positive integer in the *id_range* of a Component_Group element. The attribute *type* of the Component_Group element must be *remote*. An initialization element declares how to initialize a component attribute. It contains two attributes, *name* and *value*, where *name* is the name of the attribute, and *value* is the initial value of the attribute. The Value element contains the value of the element defined in the Name element. The value must be completely specified. If the attribute represents a group of values, such as an array or object attribute, the values must be grouped in "()".

3.2.1.2.4.2 The MOM Element

The MOM element includes a set of ptpChannel elements (point-to-point message channels) and a set of psChannel elements (publisher/subscriber distributed event channels). These channels can be accessed by any component by referring to its name.

Each element defines a channel.

A ptpChannel element contains three attributes, *name*, *buffer_size* and *message_datatype*. The value of *name* is the name of the channel. It must be unique. The *buffer_size* attribute defines the number of messages the channel can hold. The buffer size should be set to minimal to reduce verification complexity. The default value is 3. The attribute *message_datatype* declares the datatype of the messages passing through the channel. In our model, a channel can only carry messages of a certain type. The value of *message_datatype* can be of a default datatype, namely *int*, *short*, *byte*, *bool* or *bit*, or it can be a *Data* stereotyped class defined in a UML class diagram. A single dimension array of these datatypes may also be used. A psChannel element has an additional attribute: *max_subscriber*. This attribute sets the maximal number of subscribers the channel can have. The default value is 2.

3.2.1.2.5 The Process_Deployment Element

The Process_Deployment element contains one or more Process element. Each Process element indicates that such a process should be created. In our model, a process may own a set of local components and/or a set of POAs. A POA manages a set of remote objects (See 3.3.3.4.1, The CORBA Element). Unless a process is a pure remote server, in which case it contains only the POAs, the execution of the process starts with the *main* method, which resides in a *Main* stereotyped class instance.

In our model, the execution threads within a process may use local signal events to communicate with each other. A Process must also set up the signal events it used.

Correspondingly, a Process element in the deployment file may contain a Main_Class element, a Local_Components element, a POA_IDs element and a set of Signal element. The Local_Components element cannot exist without a Main_Class element. A Process element also contains an attribute named *id*, which takes the following form:

process_n

where n is a positive integer. The value of *id* should be unique. It identifies a process from others.

The Main_Class element contains an attribute named *class_name*. The value of the attribute must be the name of a *Main* stereotyped class defined in UML class diagrams in our model. We use the Main_Class element to create an object instance of that class. This object cannot be initialized, and the execution of the process starts with its main method.

The Local_Components element contains one or more Component elements, which are similar to those in POA elements (See 3.3.3.4.1, The CORBA Element). The only difference is that the *type* value of the corresponding Component_Group element should be *local* instead of *remote*.

The value of the POA_IDs element is a list of POA ids, which identifies the process as the owner of these POAs. A POA must be owned by one and only one process

Each Signal element sets up a type of signal event. It includes two attributes. The first is *type*, which is a *Signal* stereotyped class name defined in UML class diagrams. The second one is *max_subscriber*, which defines the maximum number of listeners allowed for that signal event within the process. There should not be more than *max_subscriber* number of execution threads acting as the receptors of the signal event in the process.

3.2.1.2.6 A Deployment File Example

We conclude the introduction to the deployment file by giving an example. The UML class diagram used in the example of the deployment file is defined in Figure 3.8 in section 3.2.3.2.2. The deployment file declares a distributed system with two processes. The system uses a middleware that supports both DOS and message-oriented communication. The CORBA ORB is a multithreaded one, and the buffer size for distributed message channels is 6. When the channel overflows, the new message will be blocked. Three local components of interface type "CI" are declared and deployed to the two processes. Three remote object components with interface type *RI* are declared and deployed to two POAs. Both POAs are owned by the same process. There are two distributed message channels: one is a publish/subscribe channel, the other a ptp (point-to-point) channel. The second process uses signal event *S*, which is local and there can be at most two reception threads for *S* in the process (which will be introduced in section 4.4.2).

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE Deployment system "c:\mixup\deployment.dtd">
<Deployment>
 <Component_Set>
   <Component_Group type = "local"   interface = "CI" id_range = 3_5>
   <Component_Group type = "remote" interface = "RI" id_range = 10_12>
 </Component_Set>

 <Middleware>
   <CORBA>
     <ORB id = "1">
       <POA id = "p_0" thread_policy = "thread_per_object">
        <Component id = "10">
          </Component>
          <Component id = "11">
            <Initialization>
               <Attribute attName = "a" value = "5">
            </Initialization>
          </Component>
        </POA>

        <POA poa_id = "p_1" poa_thread_policy = "thread_per_object">
          <Component id = "12"> </Component>
        </POA>
      </ORB>
    </CORBA>

    <MOM channel_size = "6">
      <psChannel name = "eventC" buffer_size = 3 message_datatype = "MT" max_subscriber = "3">
      <ptpChannel name = "ptpC" buffer_size = 4 message_datatype = "int[3]">
    </MOM>
  </Middleware>

  <Process_Deployment>
    <Process id = "process_1">
        <Main_Class class_name = "MC1">
        <Local_Components>
            <Component id = "4" > </Component>
            <Component id = "3">
                <Initialization>
                  <Attribute name = "c" value = "(3 4 5)">
                </Initialziation>
            </Component>
        </Local_Components>
    </Process>

     <Process id = "process_2">
        <Main_Class class_name = "MC2">
        <Local_Components>
            <Component id =  "5" > </Component>
        </Local_Components>
        <POA_IDs> p_0 p_1</POA_IDs>
        <Signal type = "S" max_subscriber = "2" ></Signal>
     </Process>
  </Process_Deployment>
</Deployment>
```

Figure 3.4 A deployment file example

As we can see, the deployment file is a valid XML document. It is well structured, precise and complete. The deployment file can be easily updated, especially when we change the configuration of the middlewares.

3.2.2 IPC Client Interfaces

As we mentioned at the beginning of section 3.2, in our model, interprocess communication is handled by objects of six special IPC client interface classes, namely *Stub, Response, Publisher, Subscriber, Sender* and *Receiver*. The interface instances can be used as class attributes, as method parameters or as local variables within a method. Before introducing the syntax and semantics of UML class diagrams and UML statechart diagrams, we define these six special interface types for issuing IPC requests.

We defined a UML package named "IPC utility" which includes the class diagrams that define the six interface types. The class diagram is shown in figure 3.5. These interfaces are of a special interface stereotype, named *IPC,* which is not shown on the diagram.

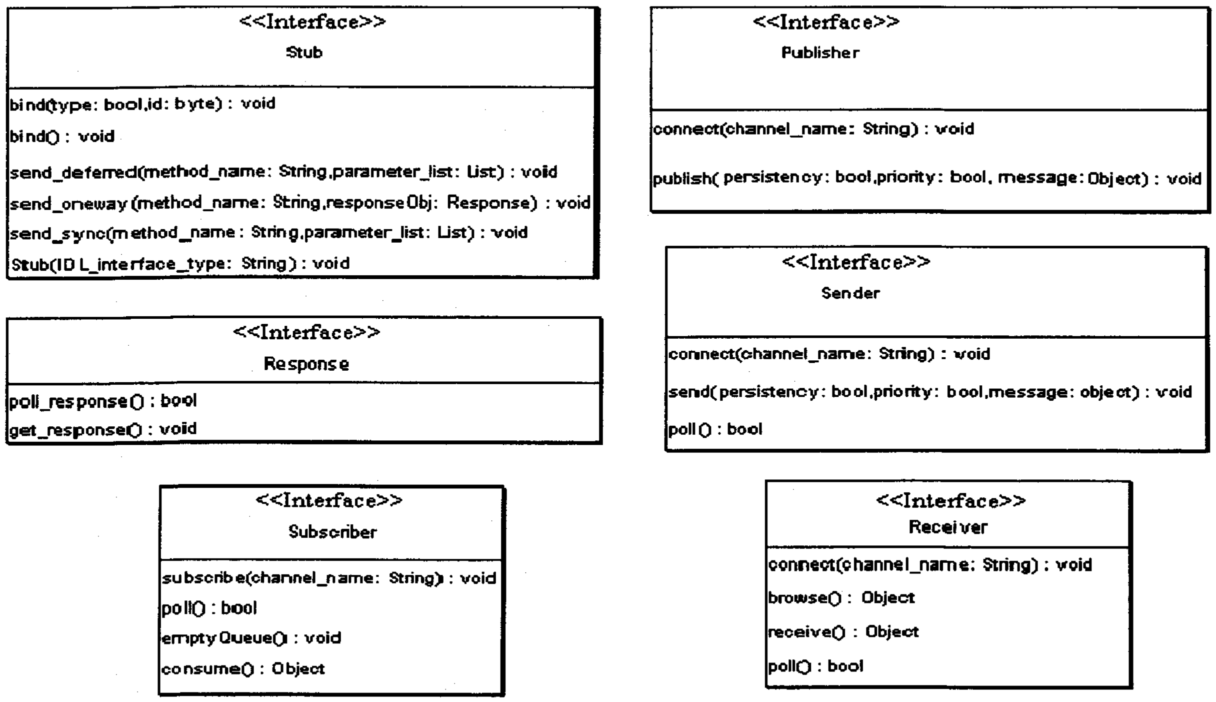

<<Interface>>
Stub
bind(type: bool,id: byte) : void
bind() : void
send_deferred(method_name: String,parameter_list: List) : void
send_oneway(method_name: String,responseObj: Response) : void
send_sync(method_name: String,parameter_list: List) : void
Stub(IDL_interface_type: String) : void
<<Interface>>
Publisher
connect(channel_name: String) : void
publish(persistency: bool,priority: bool, message: Object) : void
<<Interface>>
Sender
connect(channel_name: String) : void
send(persistency: bool,priority: bool,message: object) : void
poll() : bool
<<Interface>>
Response
poll_response() : bool
get_response() : void
<<Interface>>
Subscriber
subscribe(channel_name: String) : void
poll() : bool
emptyQueue() : void
consume() : Object
<<Interface>>
Receiver
connect(channel_name: String) : void
browse() : Object
receive() : Object
poll() : bool

Figure 3.5 IPC Interfaces

In our model, the *IPC* interface stereotype is reserved for the above interface types. Since most developers prefer treating middlewares as black boxes, the behavior of these interfaces is not defined by UML statechart diagrams. Instead, they will be incorporated directly into the PROMELA code by the translator.

In the rest of this section, we explain the functionality of each IPC interface type.

3.2.2.1 The Stub and the Response Interface Instances

A Stub instance is a client proxy of a remote object. A Response instance is used solely in method *send_deferred* of a Stub instance. They form the client-side CORBA interface

3.2.2.1.1 Initialization and Connection

In our model, a Stub instance can only access remote objects of a certain type. It is always initialized with an IDL interface name, which is actually the type of the remote object it can access. For example, the following statement creates a Stub instance *stubObj* that can access a remote object with IDL interface *RI*:

```
Stub stubObj = new Stub("RI");
```

For simplicity, in our model, a Stub instance can be connected to only one remote object. If the Stub instance is used as class attribute, the type of the remote object must be provided as the initial value of the Stub instance. Otherwise, the Stub instance is assigned to a type of a remote object by the above statement. The connection to an actual remote object is achieved through the *bind* method. We do not allow *rebind* in our model.

The *bind* method may be called without any parameter. In this case, the Stub instance is connected to any of the remote objects with a suitable IDL interface. Currently, we do not assign priority to the suitable remote objects, regardless of their location, thread policy, etc.

The *bind* method may also be called with two parameters. The first parameter is a boolean flag. It identifies the second parameter as either a POA id (when the flag has value 0 or constant POA_ID) or a remote object id (when the flag has value 1 or constant OBJ_ID). In the former case, the Stub instance is randomly connected to a remote object with a suitable IDL interface in that POA. In the latter case, the Stub instance is connected directly to the remote object with that particular id. For example, the following statement connects the Stub instance *stubObj* to the remote object with id *11*, which is defined in the deployment file in figure 3.4 as a remote object with interface *RI*:

```
stubObj.bind(OBJ_ID, “11”)
```

On the other hand, the statement:

```
stubObj.bind(POA_ID, “p_0”)
```

connects *stubObj* to either remote object *10* or remote object *11*.

For simplicity, we do not consider disconnecting a Stub instance from a remote object.

3.2.2.1.2 Remote Method Calls

After a Stub instance is bound to a remote object, service requests can be sent to the remote object through the Stub instance.

We model remote method calls of all the three synchronization styles standardized in CORBA.

- A remote method can be called synchronously: the caller halts its execution until the remote call completes.
- A remote method can be called with deferred synchronization: the caller continues its execution after the remote methods call, and polls the result of the call later.
- For a remote method without returning parameters, it can also be called asynchronously: the caller continues execution after issuing the call.

A restriction we put on all remote methods is that they cannot have a return value. This is due to the nature of Promela. If needed, a return value can be simulated as a special output parameter (as denoted by *out* in some programming languages such as Ada).

A *synchronized remote method call* is performed through the *send_sync* method. The method takes the name of the remote method and a list of the method's parameters as its input parameter. For example, suppose a method on IDL interface *RI* is defined as the following:

```
rm(int p1, int[3] p2) : void
```

where p1 is an *in/out* parameter and p2 is an *in* parameter. The remote method can be called in a statechart diagram as:

```
stubObj.send_sync("rm", v1, v2)
```

where stubObj is a Stub instance, v1 is an integer attribute and v2 is an integer array attribute with three elements.

An *asynchronous remote method call* is defined analogously through the *send_oneway* method.

A *deferred remote method call* is a bit more complicated. It is performed through the *send_deferred* method. Before the method can be called, a Response object must first be defined. The response object will be sent together with the remote method name and remote method parameters. For example, the following statement calls the method *rm* with deferred synchronization:

```
stubObj.send_deferred(rm, response, v1, v2)
```

Here, *response* is a pre-defined Response instance. A Response instance is always associated with a method in a remote object. The caller continues its execution after having issued a deferred call and later on, calls either the *poll_response*() or the *get_response*() methods of the response object. The method *poll_response()* returns 0 if the remote call has completed and 1 otherwise. In either case, the call returns immediately. The *get_response()* method, on the other hand, will be blocked until the remote call completes. When the method returns, all output parameters will be updated. In the above example, the value of *v1* will be updated when the method *get_response*() returns. For more details, please refer to Figure 6.3 in section 6.1.

3.2.2.1.3 The Short Cut for Remote Attribute Access

As we will explain in section 3.2.3, a remote attribute *attr* is exposed on IDL interface with two functions, "set_attr" and "get_attr". If "set_attr" is omitted, the attribute is read-only.
For simplicity, we allow the following statements in the UML statechart diagrams:

```
stubObj.attr = v
```

which assigns value *v* to attribute *attr*. The statement is equal to:

```
stubObj.send_sync("set_attr", v)
```

On the other hand, the statement:

```
p = stubObj.attr
```

assigns the value of *attr* to *p*. The statement is equal to:

```
stubObj.send_sync("get_attr", p)
```

3.2.2.2 Publisher and Subscriber Instances

A distributed event is a message stored in a distributed event channel as defined in the deployment file. The Publisher instances publish distributed events and the Subscriber instances receive distributed events.

A Publisher instance contains two methods: *connect* and *publish*. Method *connect* connects the object to a distributed event channel. Suppose a distributed event channel defined in the deployment file is called eventC, the following statement connects the Publisher instance *publisher* to *eventC.*

```
publisher.connect("eventC")
```
For simplicity, we do not allow reconnection or disconnection in our model.

Method *publish* publishes an event by sending it to the distributed event channel. The priority of the event message can be specified as either *urgent* or *normal.* The persistency of the message can be specified as either *persistent* or *transient.* An urgent message may be retrieved before a normal message sent before it. A *persistent* message must be retrieved by all the subscribers. It might block the publisher if the channel is full. A *transient* message may be lost if the channel is full. A transient message will never block the publisher. *urgent*, *normal*, *persistent*, *transient* are constants used in our model: urgent = persistent = 1 and normal = transient = 0.

The message itself can be an attribute of a default datatype, or be specified as an object, with its class definition given in the class diagrams. For convenience, the method can be called differently in the statechart diagrams: the object can be substituted by a list of parameters, each representing an attribute in the object. For example, in Figure 3.8 class *MT* is specified with two attributes, namely a boolean flag and an integer. In the deployment file shown on Figure 3.4, the message type in the distributed event channel *eventC* is defined as "*MT*". Suppose a Publisher instance called *publisher* is already connected to the channel *evnetC*. Then the following two cases are equivalent in our model:

case1: MT mt = new MT(FALSE, 10);   publisher.publish(normal, transient, mt)
case2: publisher.publish(0,0, 0, 10)

Note that FALSE is another constant, with FALSE = 0. The opposite is the constant TRUE, with TRUE = 1.

The order of distributed events published by the same publisher will be reserved. However, distributed events published by different publishers might be retrieved in an order different from the one they were published.

A Subscriber instance receives distributed events. It contains four methods: *subscribe, emptyQueue, consume* and *poll.*

Method *subscribe* is used to connect the Subscriber instance to a distributed event channel. In reality, the method creates a hidden message queue, which is accessible only by the *Subscriber instance.* The queue will receive a copy of each message sent to the distributed event channel after the queue is connected to it. When the queue is full, new message coming to the queue will be dropped.

Suppose a Subscriber instance is called *subscriber* and the distributed event channel it wishes to receive messages from is *eventC*, as defined in Figure 3.4. Then the statement:

    subscriber.subscribe("eventC")

will connect object *subscriber* to channel *eventC.*

In our model, method *subscribe* can only be called once for simplicity. The message queue can be forcefully emptied by calling method *emptyQueue*, when necessary.

Method *consume* retrieves and returns the oldest message in the message queue. If the queue is empty, the method call is blocked. Similar to the *publish* method defined for Publisher instances, method *consume* may return a list of object attributes instead of a single object. For example, the following two sets of statements are equivalent:

```
bool flag; int k; Mt mt; mt = subcriber.consume(); flag = mt.flag; k = mt.data
bool flag; int k;        (flag, k) = subscriber.consume()
```

Method *poll* is a non-blocking call. If the message queue is empty, the method returns 0, otherwise it returns 1.

3.2.2.3 Sender and Receiver instances

A Sender instance sends messages to a ptp (point-to-point) message channel. It is similar to a Publisher instance.

The method *connect* in a Sender instance is similar to the one in a Publisher instance. The method *send* in a Sender instance is similar to the method *publish* in a Publisher instance. The only difference is the message will not be broadcasted to its receivers as the distributed events were to their subscribers.

A Receiver instance receives messages from a ptp message channel. It is similar to a Subscriber instance. The syntax for method *poll* is the same in both classes. Instead of method *consume* in a Subscriber instance, we have two methods *receive* and *browse* in a Receiver instance. A message in a point-to-point message channel is not broadcasted. If more than one Receiver instances wish to read the same message, method *browse* instead of *receive* should be called. The former reads the message without removing it from the channel, while the latter will read and remove the message. The method *connect* connects the Receiver instance to a point-to-point message channel, similar to the one in a Sender instance.

3.2.2.4 Conclusion

In section 3.2.2, we introduced the six IPC client interface classes we defined. They represent the IPC client interfaces provided by the middlewares we introduced in Chapter 2. By incorporating these interface classes into our design model and their behavior into the verification model accordingly, problems caused by resource limit, event-reordering phenomena, etc. can be caught during the verification.

3.2.3 Syntax of the Restricted UML Diagrams

As we explained at the beginning of this Chapter, a pre-condition for any automated translation of UML models is that the designer must design concrete and unambiguous UML diagrams. The UML class diagrams and statechart diagrams used in our model are no exception. The syntax of these diagrams must be restricted to remove ambiguity and incompleteness. Another reason to restrict the UML diagrams is to cope with the expressiveness power of PROMELA language. The expressiveness of PROMELA language is stronger than most other formal specification techniques, but still, it is much more restrictive compare to general UML notations.

The most rigid restriction we put on the UML diagrams is the range of datatype allowed in the model, which will be explained below. Other restrictions concerning UML class diagrams will be explained in section 3.4.2. The restrictions concerning UML class diagrams will be explained in section 3.4.3. There is no guarantee the translator will catch the violation on these restrictions. The translator may abort or produce problematic PROMELA code with or without warning.

3.2.3.1 Datatypes and Constants

In our model, basic datatype includes only integer and subset of integers. More precisely, the basic datatypes are:

- *int*: An integer. (-2^31 - 1 .. 2^31 - 1)
- *short*: A short integer. (-2^15 - 1 .. 2^15 - 1)
- *byte*: Value 0..255.
- *bool:* Value 0 or 1
- *bit:* Value 0 or 1

Customized datatypes can be constructed from the above basic datatypes by declaring in UML class diagrams *Datatype* stereotyped classes that contain no method definitions. Classes without a stereotype may also be used as datatypes, but not always. We forbid using a non-stereotyped class as datatype unless it is for a class attribute of another non-stereotyped class. The role of different class stereotypes in our model will be explained in the next section.

For multiple-data declaration, we only support fixed-size single-dimension arrays. In other words, the *multiplicity* value of any class attribute must be *n..n* or *n*, where *n* is a positive integer. In addition, an array cannot be used as method parameter. To get around with these restrictions, users can define Datatype stereotyped class with array attributes.

While these restrictions seem to us rather limited at the first glance, they are sufficient enough to specify many abstract design models. In fact, they are much more flexible than the datatypes provided by most other formal specification languages. Some languages, for instance, only allow datatype defined through enumeration.

For clarity, we define a set of constants in our model:

Constant Name	Constant Value	Situation
TRUE	1	used as a boolean expression
FALSE	0	used as a boolean expression
urgent	1	used as message priority
normal	0	used as message priority
persistent	1	used as message persistency flag
normal	0	used as message persistency flag
OBJ_ID	1	used as the binding option for Stub

		instances
POA_ID	0	used as the binding option for Stub instances

Figure 3.6 Constants

3.2.3.2 UML Class Diagrams

In our model, UML class diagrams provide the building blocks of a distributed system: The building blocks are classes. In this section, we explain different class stereotypes and the relationships between classes.

3.2.3.2.1 Class Stereotypes and Interface Stereotypes

Different stereotyped classes or interfaces play different roles in our model. We already introduced the IPC interface stereotype in section 3.2.2, which is reserved for the six pre-defined IPC client interfaces introduced in the same section. In addition to the IPC stereotype, we defined five other class and interface stereotypes, namely *Component_Interface*, *IDL_Interface*, *Main*, *Signal* and *Datatype*. The former two are interface stereotypes and the latter three are class stereotypes. An interface with stereotype *Component_Interface* is a local component interface. Local components in our setting are specified in the deployment file using such interfaces. A class with stereotype *IDL_Interface* is a remote interface. Remote objects are specified in the deployment file using such classes. Classes with stereotype *Main* are used in the deployment file for the instantiation of the object with the main execution thread of a process. A class with stereotype *Signal* is a local signal-event type. A class with stereotype *Datatype* is used solely as a customized datatype. No other class stereotype may be defined in our model. The syntax and semantics of the five class stereotypes are summarized in Figure 3.5:

Class|Interface Stereotype	**Syntax and Semantics**
Component_Interface	A component interface that is directly accessible within the process boundary. The deployment file uses interfaces of this stereotype to create local components.
IDL_Interface	A remote object interface, which is accessible application-wide through IPC interfaces. The deployment file uses interfaes of this stereotype to create remote object components.
Main	A class owns a *main* method. The method has no input or output parameters. The *main* method is reserved for *Main* stereotyped class only. The deployment creates the *Main_Class* element of a process from these classes.
Datatype	A class without any method. The class is used solely as attribute datatype. In our model, any class without methods must be

	defined as a *Datatype* class.
Signal	A class without any method. It represents a local signal event used in statechart diagrams.

Figure 3.7 Class stereotypes

In addition to classes of these five stereotypes, users may also define classes without any stereotype. In the next section, we present the relationships between the classes and interfaces in UML class diagrams.

3.2.3.2.2 Relationships between Classes/Interfaces

The translator will ignore any relationship between classes/interfaces defined in the class diagrams other than generalization and realization. Relationships such as composition and association among classes, for example, is allowed but ignored by the translator, since they will not affect the behavior of the system specified in UML statechart diagrams.

When generalization relationship exists between two classes, the subclass will include the class attributes and the methods defined in the super class. We do not support attribute overload or method overload for super- and sub- classes. This means if a class attribute or a class method is defined in the super class, it cannot be re-defined in the subclass. Multiple inheritance is supported. Note that the order of the class attributes is important to the Signal class and the Datatype classes used as message types for distributed channels. To solve the order confusion caused by multiple inheritance, we suppose in such classes, the class attributes are organized in the alphabet order.

In our model, generalization relationship can exist between two classes if and only if one of the following rules applies:

- The classes are of the same stereotype
- The super class is a "Datatype" class.
- The super class is a non-stereotype class and the subclass is not a Datatype or Signal stereotype class. We add this restriction because a non-stereotype class in our model always contains methods, while Datatype or Signal stereotype class does not allow class methods.

Since the IDL interfaces and the Local interfaces will be used to define components in our model, for simplicity we forbid a class realizing an interface of local or IDL to be used as super class.

In our model, realization relationship is reserved between a Component_Interface stereotyped interface or an IDL_Interface stereotyped interface and a non-stereotype class. No other type of interface may be defined in our model. All methods in the former must be implemented by the latter.

Special methods are those starting with *set_* and *get_*. Such methods are reserved for the read/write of class attributes (See section 3.2.2.2). Suppose the interface has a method named *get_attr(obj: C)*, where *obj* is an output parameter, then the class realizing the interface must have a *C* type class attribute named *attr*. If *set_attr(obj:C)*, where *obj* is an input parameter, is not defined in the interface, the attribute will be a read-only attribute. As a short cut, such attributes can be referred to directly in the statechart diagrams instead of using the *set_* and *get_* functions. (See section 3.2.2.2)

No return value is allowed for any methods in our model. A return value, when needed, must be substituted by an *output* parameter, as the case of the *get_* methods.

Figure 3.8 shows a UML class diagram example.

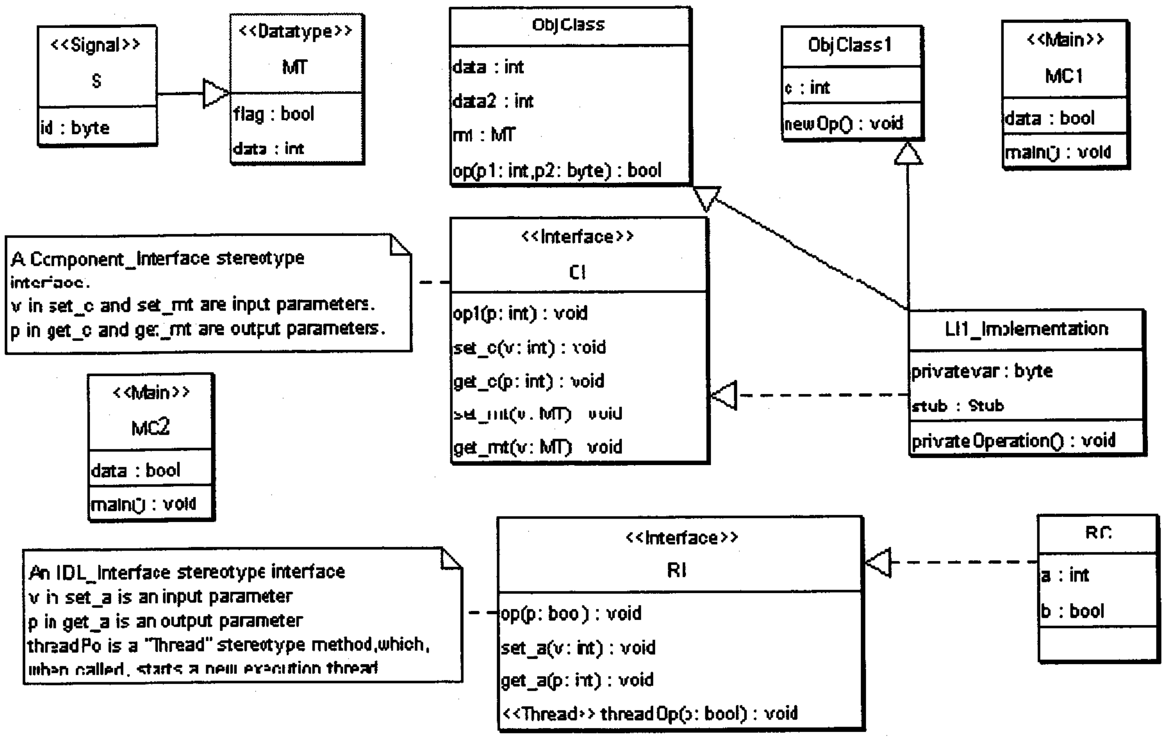


Figure 3.8 A class diagram

Note that in the current stage, our translator does not check visibility violations. In other words, the translator treats all attributes as public. For example, if a private attribute is accessed from outside of the object, the translator will not complain.

The behavior of non-stereotyped classes used in UML class diagrams is defined by UML statechart diagrams. In our model, except for the *set_* and *get_* methods, each class method is described by a statechart diagram.

3.2.3.3 UML Statechart Diagrams

UML statechart diagrams represent the behavior of entities capable of dynamic behavior by specifying its response to the receipt of event instances [17]. In our model, UML statechart diagrams describe the behavior of the classes, including the correctness criteria of the behavior. Each class method, except the set_ and the get_ methods, is described by a statechart diagram. The behavior of a class is thus defined by the collection of statechart diagrams corresponding to its methods.

A UML statechart diagram is a graph consists of states (notes) and transitions (arcs). In this section, we regulate the syntax of UML statechart diagrams so that they can be recognized by a translator. First, we explain the restrictions on states. Second, we explain the restrictions on transitions. Then we explain how concurrency and synchronization are modeled.

3.2.3.3.1 Restriction on States

A state is a condition or situation during the life of an object in which it satisfies some particular condition, performs some particular activity, or waits for an event. In UML statechart diagrams, a state is denoted as a round rectangle with a *name* section and an *internal action* section. UML provides several state types. We forbid *history*, *sync*, *fork* and *join* states in our model, and composite states in our model cannot be split into concurrent regions. In other words, there will be no concurrency element in statechart diagrams, except the call to a *Thread* stereotype method, which starts a new execution thread in the statechart diagram. Sub-state machine is another feature omitted.

The state name needs not to be unique. In our model, a state with name starts with *end*, *progress*, *accept*, *atomic* or *d_step*, is special:

- A state whose name starts with *end* is a state that is acceptable as a valid termination point of the process execution. If a process stops at a state other than a state with name starting with *end* or its end states, SPIN will report a deadlock violation during verification or simulation.
- A state whose name starts with *progress* must be visited infinitely often in any infinite system execution, otherwise SPIN will report error during verification or simulation.
- A state whose name starts with *accept* can only be visited finitely many times in any infinite system execution, otherwise SPIN will report error during verification or simulation.
- If a composite state has a name starting with "*atomic*", the translator will interpret the internal action of the composite state as been executed atomically.

The first three are for verification purpose. They will not affect the system execution. The last two are used for synchronization purpose and for reducing complexity.

To simplify the translation process, the only internal transition of a state we allow in our model is an entry action. Actions will be introduced in more detail in section 3.2.3.3.2.

We mark the entry action belonging to the top composite state of a statechart diagram as a special action. It is the only action allowed for defining local variables. Since a statechart diagram in our model represents a method, we need to define local variables of the method. In our model, local variables must be defined in the entry action belonging to the top composite state of a statechart diagram. Multiple variable declarations will be separated by ";". For example, the following entry action defines three local variables:

```
int a, b[2]; Stub c = new Stub("RI");
```

We do not support variable overloading. The name of a method parameter or a method local variable cannot be the same as a class attribute of the class that owns the method.

3.2.3.3.2 Restriction on Transitions

A transition is shown on a UML statechart diagram as a labeled arrow between two states. The state the transition leaves is called the source state of the transition and the state the transition leads to is called the target state. If the source state is an initial state, the transition is called an *initial transition*. If the target state is an end state, the transition is called an *end transition*. If the source state is a fork state, we call the transition a *fork transition*.

The labels on a transition may contain:

- An event signature which triggers the transition when it occurs.
- A boolean expression guard which must be satisfied for the transition to *fire* when it is triggered.
- An action expression that performs some action when the transition is *fired*.

A transition is *triggered* when the event occurs, or, if it has no trigger event, when the system leaves its source state. A transition *fires* when it is triggered and the guard condition is satisfied. The action on a transition will be enabled when the transition fires. The system state transfers to the transition target state when the action completes.

Compared to other UML translation efforts [12, 13, 15, 18], our UML statechart diagrams are simple in structure (very little concurrency elements, for example) but rich in event specification. We only allow basic transitions in our model. High-level transitions (i.e. transitions initiated from the border of a composite state), complex transitions, etc. are not supported.

In our model, the restriction on trigger events is minimal. UML statechart diagrams supports four types of trigger events, namely *signal events*, *change events* (i.e. a boolean expression, the event is triggered when the expression is evaluated to true), *call events* (method calls) and *time events*. We do not support time event, as PROMELA has no way of modeling time. Event on a transition can be either a signal event or a change event. Call events are also supported, but is customized slightly. Since a statechart diagram in our model describe the behavior of a method, for simplicity, we omit the call event at the beginning of each statechart diagram. A method call action, which generates a call event, will create a statechart diagram of that method and execute it at spot, as if the method call is the entry point to a sub-statechart diagram. The special case is when the method call is to a *Thread* stereotyped method. In this case, the method is executed on a separate execution thread as if it starts from a separate outgoing transition of a fork state.

The type of signal events is defined in a UML class diagram as a *Signal* stereotyped class. Unlike other translation attempts, we support event parameters. If the event has parameters, the value of the parameters can be either stored in an attribute or discarded by using scratch variable "_" during reception. For example, in Figure 3.8, we defined a Signal stereotyped class *LE*. Then, a transition with event:

S(boolVar, _, intVar)

will cause the transition to be triggered by an S event. The S class has three variables, namely *data, flag*, and id. The class attributes are ordered by the alphabet. When the event occurs, the value of attributes intVar and boolVar will be set to the values of the corresponding event parameters. Event parameters are saved in order. The variables from the super class precede other class variables. Note that since local events are not visible beyond process boundary, the transition can only be triggered by an S event in the same process.

The change event is a boolean expression. The event will occur when the boolean expression becomes true. We require that the boolean expression be enclosed with braces"()". For example, a transition with the following event label will be triggered when *a* becomes equal to *1*:

(a ==1)

A boolean expression can also be used as a transition guard. When the expression is used as a change event, a transition is triggered whenever the expression becomes true. When it is used as a guard, however, the transition cannot fire if the expression evaluates to false in the moment the transition is triggered, even if later on the expression changes to true. Guard expressions are usually used to branch the flow.

In our model, the basic boolean expressions include:

- Reserved word: "FALSE" and "TRUE".
- Numbers: 0: false. Any number other than 0: true.
- Comparison. Compare data attribute values and constants with ">","<", "=", "!=", ">=", "<=".

Basic boolean expressions can be combined into complex boolean expression by using "&", "|" and "!".

In our model, an action (i.e. a transition action or the entry action in a state) may perform one of the following tasks:

- Assign/modify value of an attribute.
- Publish a local event.
- Call a method and wait for its completion.
- On the final transition, return values to the method caller.
- Assert the truthfulness of a boolean expression

A method may assign/modify the value of an attribute in its statechart diagram unless the attribute is owned by another process or another method. An attribute is referred to by the following rules:

- If the attribute is a local attribute of the method, or a class attribute of the object that owns the method, the attribute is referred to directly by its name.
- If the attribute belongs to an object that resides in the same component as the method but does not own the method, the attribute will be referred to by prefixing its name with the object name followed by a dot ".". For example, *studentObj.score* refers to an attribute named *score* in an object named *studentObj.*
- If the attribute belongs to an object that resides in the same process but not the same component as the method, the attribute will be referred to by prefixing its name with the interface type of the component, plus "(component ID)", the object name and ".". For example, *LI(2).studentObj.score* referes to an attribute named *score* in an object named *studentObj* in component 2, which is an *LI* type component.

To distinguish a local event from a method call, a local event must be published by the following action:

event(signalClass, p1,...pn)

where *event* is a keyword. *signalClass* is the name of a *Signal* stereotyped class. The rests are parameters. Accordingly, no method can be named *event* in our model.

A method is referred to in the same manner as an attribute. Note that interprocess communication in our model is achieved by calling methods through an IPC stereotyped interface.

If a method has no *out*, *inout* or return parameters, the final transition of its statechart diagram must be a blank transition. For a method with *i* output parameters (*i*>0). The final transitions must have no trigger, no guard. Its effect must be a "return" action. A return action takes the following format:

return(v1, ...vi)

Here keyword *return* identifies the return action. Accordingly, in our model, a method cannot be named "return". Each return value (v1 ...vi) can be either a number or an attribute name. The sequence of the return values is the same as the sequence of the method's output parameters defined in class diagrams. The *return* parameter precedes other parameters. Each return value must be compatible with the datatype of its corresponding output parameter. For example, a return value *4* is illegal if the corresponding output parameter is of datatype *bool.*

An assertion action traps violations of simple safety property during verification and simulation:

```
assert(boolean_expression)
```

In the above action, *assert* is a keyword identifying an assertion action and the parameter *boolean_expression* is a boolean expression that should hold true at that point of execution. The action is always executable. SPIN will report error if the expression is evaluated to false or 0 during verification or simulation. To avoid confusion, no method in our model can be named *assert.*

3.3 Conclusion

In this chapter, we defined a UML design model for distributed systems with middleware. We use a deployment file to set up a concrete system from classes and interfaces defined in UML class diagrams. The behavior of the UML classes is defined in UML statechart diagrams.

The UML statechart diagrams in our model uses predefined IPC client interfaces to perform interprocess communication. We minimize the concurrency elements in the UML statechart diagrams and consequently greatly simplified its structure. Our focus is on the modeling of signal events, method calls, change events and transition guards, which makes our approach unique from other attempts to translate UML diagrams. In addition, we incorporate verification elements in the UML statechart diagrams, such as assertion statement, progress and legal end states, etc., so the SPIN model checker can be used to verify the correctness of the model. At current stage, however, we do not support LTL property specification.

In the next three chapters, we'll introduce the translation of the UML design model introduced in this chapter. In chapter 4 we will introduce the elements with no relationship to interprocess communication, especially the local components and the signal event. In chapter 5 we will introduce the elements related to MOM. In chapter 6 we introduce the elements related to DOS middleware.

Chapter 4 Mapping Objects to PROMELA

In Chapter 3, we introduced our verifiable UML design model for the distributed systems with middlewares. Starting from this chapter, we show how such a model is mapped into the PROMELA verification model. Our UML design model is object-oriented. Each object contains a data area and an operation area. In this chapter, we show how to map objects.

PROMELA is not an object-oriented language and it does not support method invocation. Similar to other efforts made on translating objects of various languages into PROMELA [2 6], we choose to map the data area of an object into a PROMELA record and the operations into PROMELA inline macros. The main character of our translation is that the restriction on the syntax of the operations is minimal, although neither references nor pointers are allowed in the model.

Section 4.1 introduces the mapping for object data and section 4.2 introduces the mapping for operations, including the mapping for signal events.

4.1 Object Data Mapping

To map the data area of an object, we define a PROMELA record type for the class of the object. A record of that type is then defined to represent the data area.

4.1.1 Record Type Definition

For each class defined in the UML design model, we generate a PROMELA record type (typedef) to represent its data area, unless the data area is empty. The data area of a class may not be equal to the set of the data attributes of the class: Unless the class is a *Component_ Interface* or *IDL_Interface* stereotyped class, the data area of the class includes its own data attributes and the data attributes of its super classes. Otherwise, the data area of the class is the data area of its implementation class.

Suppose a class named "classname" has n attributes in its data area. For the ith attribute, we denote its datatype as "type_i", its name as "name_i" and its multiplicity as "mul_i". We define att_i as name_i if mul_i = 1, as name_i[mul_i] if mul_i>1.

If n>0 and the class is not a signal stereotyped class, the following PROMELA record type will be defined for the class:

```
typedef classname_Class
{
 type_1 att_1;
   .
   .
   .
 type_n att_n;
}
```

If the class is a signal stereotyped class, the following PROMELA record type will be defined for the class:

```
typedef classname_Class
{
  byte x_pid;  /*The id of the process in which the event sender and the event receiver
                 reside */
  type_1 att_1;
    .
    .
    .
  type_n att_n;
}
```

In other words, the name of the PROMELA record type is the class name suffixed with *_Class*. For each data attribute in the data area of the class, a data field with the same name and datatype is added to the PROMELA record type. The multiplicity of the data attribute is translated into array size.

With the record type definition in place, we can define a record to represent the data area of an object.

4.1.2 A Record Representing an Object Data Area

There are two types of objects in our model. The first is the component objects statically defined in the deployment file. The second is the data objects dynamically created in the execution of the methods.

A component object must be a instance of a *Component_Interface*, *Active_Object* or *IDL_Interface* stereotyped class. Each component object is given a unique id. The component objects of the same class type are defined as a group and given continuous ids. For example, the following statement in the deployment file defines three component objects, the class type is *LI1*, and the ids are *3, 4, 5*, respectively.

```
<Component_Group type = "local"  interface = "LI1" id_range = 3_5>
```

To represent the data areas of such objects, we define a global record array for each component group defined in the deployment file:

Let us denote the name of a component group class as "classname", the id range of the component group as low_high. Let arraysize = high-low+1. Then the global record array is defined as:

classname_Class classname_obj[arraysize];

The ith element in the array represents the data area of the object low+i.

A data object, on the other hand, must be an instance of a *Data* stereotyped class. Such an object has no operations. They can be used as local attributes of a method or can be passed as input parameters of a method in a value-passing manner.

The mapping of the data objects is straightforward: the name of a data object variable remains unchanged, while the variable datatype changes from the class name to the corresponding PROMELA record type (c.f. section 4.1.1).

4.1.3 Related Macros and Inlines

For clarity and convenience, we define a set of PROMELA macros and inlines to simplify the read/write of object data. For each record type (typedef) in our verification model, we define a *copy* inline and a *clear* inline. In addition, for each component object class, we define a *getObj* macro.

The reason for us to define the *copy* and the *clear* inlines is that the typedef structure in PROMELA is rather limited. It cannot be used directly in value assignments: the only way to copy a PROMELA record to another is to copy each and every record attribute separately. For example, the statements below will result in a compiling error, when a, b are both PROMELA records:

```
A_Class a, b;
a.attrA = 5;
b = a; /* Error!  Correction: b.attrA = 5; or x_assign_A(b, a) */
```

To solve the problem, we define a *copy* inline and a *clear* inline for each record type. The role of the *copy* inline is to copy the value of a record to another and the role of the *clear* inline is to reset the values of all data attributes in a record to *0*. To reduce the complexity of the verification, the copying and the clearing are performed atomically without interleaving.

1. The "copy" inline:

For each record type named "RecordType", we define an inline "x_copy_RecordType(x_a, x_b)". X_a, x_b should be variables of the record type The algorithm for the inline is as follows:

```
for each data field "f" in the record type, let's denote its datatype as FType
 if f is not an array,
   if FType is primitive, x_a.f = x_b.f;
   else x_copy_FType(x_a.f, x_b.f),
   end if;
 else
    for each element in array f,
      if FType is primitive,
     x_a.f[j] = x_b.f[j],
     else x_copy_FType(x_a.f[j], x_b.f[j],
      end if
    end for
  end if
end for
```

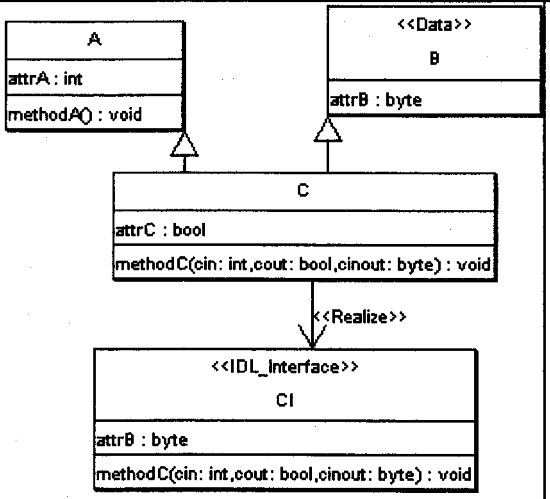
2. The "clear" inline:

For each record type named "RecordType", we define an inline "x_clear_RecordType(x_a)". The input parameter a should be a variable of the record type. The algorithm for the inline is as follows:

```
for each data field "f" in the record type, let's denote its datatype as "FType"
  if f is not an array,
    if FType is primitive , x_a.f = 0;
    else x_clear_FType(x_a.f),
    end if;
  else
    for each element in array f,
      if FType is primitive ,
        x_a.f[j] = 0,
      else x_clear_FType(x_a.f[j]),
      end if
    end for
  end if
end for
```

For example, the following table shows the data area mapping for classes A, B, C and CI. Since the class A and the class B are both super class for the class C, the record type for the class C includes the data attributes in the class A and the class B. Since the class C implements the class CI, the data area mapping for CI and C is the same.

UML Class Diagram	PROMELA Records	Inlines

 A
attrA : int
methodA() : void
<<Data>>
B
attrB : byte
C
attrC : bool
methodC(cin: int,cout: bool,cinout: byte) : void
<<Realize>>
<<IDL_Interface>>
CI
attrB : byte
methodC(cin: int,cout: bool,cinout: byte) : void
Multiplicity for attrA in class A is 3...3
	typedef A_Class
	{ int attrA[3]; }
	
	typedef B_Class
	{ byte attrB; }
	
	typedef C_Class
	{
	int attrA[3];
	byte attrB;
	bool attrC;
	}
	
	typedef CI_Class
	{
	int attrA[3];
	byte attrB;
	bool attrC;
	}
		inline x_copy_A_Class(x_a, x_b)
		{ atomic{
		x_a.attrA[0] = x_b.attrA[0];
		x_a.attrA[1] = x_b.attrA[1];
		x_a.attrA[2] = x_b.attrA[2];
		} }
		
		inline x_clear_A_Class(x_a)
		{ atomic{
		x_a.attrA[0] =0; x_a.attrA[1] =0;
		x_a.attrA[2] = 0;
		} }
		
		inline x_copy_B_Class(x_a, x_b)
		{ atomic { x_a.attrB = x_b.attrB;
		}}
		
		inline x_clear_B_Class(x_a)
		{ atomic{ x_a.attrB =0; } }

*In the method methodC of the class C:
- cin is an input parameter
- cout is an output parameter
- cinout is an input/output parameter
 	
*_Deployment file fragment:_
<Component_Group type = “local” interface = “CI” id_range = 3_7>
		inline x_copy_C_Class(x_a, x_b)
		{ atomic{
		x_a.attrC = x_b.attrC;
		x_copy_A_Class(x_a, x_b);
		x_copy_B_Class(x_a,x_b);
		}}
		
		inline x_clear_C_Class(x_a)
		{atomic{x_a.attrC =0;
		x_clear_A_Class(x_a);
		x _clear_B_Class(x_b); }}

Figure 4.1 Data Area Mapping

For each component object group defined in the deployment file, we define a *getObj* macro:

> *For each component group, suppose its class name is "classname", the id range of the group is low_high. We define the following macro:*
>
> *#define x_classname_getObj(j) classname_obj[j-low] /*low<=j<=high*/*

The macro takes a component object id as input parameter and returns the record representing its data area.

4.2 The Mapping for Object Operation Area

The operation area of an object includes a set of methods. In our model, each method is assigned with a statechart diagram to define its behavior. In this section, we introduce the mapping for methods in local objects. The mapping for remote methods (methods that belong to remote objects) will be introduced in the next chapter.

This section is organized as follows: In section 4.2.1, we list the problems in expressing methods and method calls in PROMELA. In section 4.2.2, we explain the method types in our model. In section 4.2.3 we outline the mapping of methods and method calls. In section 4.2.4 we introduce the mapping for statechart diagrams in detail.

4.2.1 Problems with Expressing Methods and Method Calls in PROMELA

The concepts of method and method call do not exist in PROMELA, since PROMELA does not support functions. Instead, a PROMELA program consists of a set of *processes* running concurrently. A PROMELA process can communicate with other processes through communication channels or global variables. It may also create another process, both of which will then run concurrently.

It is natural to model an active method (i.e. a method that executes on its own execution thread) as a PROMELA proctype and a call to an active method as a *run* statement that creates a process instance from the proctype. However, there are several limitations:

- A PROMELA process does not return a value.
- The input parameters are always passed by value: The notation of the pointer or the reference does not exist in PROMELA.

On the other hand, using *PROMELA processes* to simulate *passive methods* (i.e. methods that must to be executed by others and using their caller's execution thread) is possible [8] but too expensive. This is because the number of processes in the concurrent execution is limited to 256 during verification for SPIN, and each call to a passive method makes use of a separate process.

For passive methods, a practical solution is to use PROMELA inline construct, which is a macro expansion, to simulate them. A PROMELA inline is a block of code that will be *cut-and-copy* to where it is called. Since it is a macro expansion, PROMELA inline is very efficient. However, as a substitute for methods, it has several limitations:

- inline commands do not return values.
- recursion is forbidden.
- unlike functions, the internal structure of an inline construct is visible to its caller. For example, a variable defined in a PROMELA inline will become a local variable of the calling process. If the same inline is called twice in a process, and a variable is defined in the inline, a syntax error will result since the variable is defined twice. On the other hand, any change made to a parameter passed to an inline will be made to the parameter itself in a pass-by-reference manner, instead of a copy of the parameter.

4.2.2 Method Types in Our Model

In our model, we use three kinds of methods, namely the *main* methods, the *thread* methods and the *passive* methods. The former two are variations of active methods.

- A *main* method can only be defined in an Active_Object stereotyped class. It is defined as a method named *main*, without any parameters. A main method is self-executed. Each process in our model may own an active object, the main method in which serves as the start point of the process.

- A *thread* method is a method whose name ends with *_thread*. A *thread* method must be called by other methods, but they run on separate threads. A *thread* method may take input parameters, but it will not have any output parameters since it will execute concurrently and independently with its caller. A *thread* method can be recursively called but not infinitely.

- Any other methods are *passive* methods. They must be called and executed on the same thread as its caller, that is, the caller will be blocked until the passive method finishes. A passive method can have input, output or input/output parameters, but no return value. If needed, an output parameter can be used to hold the return value. Since PROMELA inline does not support recursion, we forbid recursive calls to passive methods.

In our design model, all method parameters are passed-by-value. However, we do allow the input/output parameters. The value of such a parameter will be updated when the method call is finished. Unless the parameter needs to be accessed concurrently by methods on different execution threads, declaring it as an input/output parameter will have the same effect as passing it by reference.

4.2.3 Mapping Methods and Method calls

In the PROMELA verification model, we map active (i.e. main or thread) methods into PROMELA proctypes and passive methods into PROMELA inlines.

4.2.3.1 The Mapping for Main Methods

A *main* method will be mapped into a PROMELA proctype. For each Active_Object class, suppose its name is *cn*, the following PROMELA proctype will be generated as the mapping for its main method:

```
proctype main_cn(byte x_objID) /* each call should pass an object id */
{
  X_main_cn_locals    x_local; /* variables local to the method. See section 4.2.3.3.1 */
  ...
}
```

For each active object in the design model, the PROMELA *init* process creates a process instance from its *main* method proctype.

Suppose in the deployment file, the *id* range for the component group is *low_ high*. For low<=j<=high, the following process will be created in the PROMELA *init* process:

```
run main_cn(j);
```

4.2.3.2 The Mapping for Thread Methods

A *thread* method will also be mapped into a PROMELA proctype. However, unlike for *main* methods, no process instance will be created by the *init* process. An instance of the proctype will be created when the method is called.

For each *thread* method in our model, let us denote its name as *methodNameThread* and the name of the class who owns the method as *className.* Suppose the method has N input parameters and we denote the datatype for the Jth parameter as *typeJ* and the name for the Jth parameter as pJ. The following PROMELA proctype will be generated as the mapping for the method:

```
proctype methodNameThread_className(byte x_objID, type1 p1, ... typeN pN)
{
  X_methodNameThread_className_locals    x_local; /* see section 4.2.3.3.1 */
  ...
}
```
where x_objID is the object id.

Accordingly, a call to the *thread* method in statechart diagrams will be mapped into the creation of a process instance in PROMELA:

Thread Method Call in a Statechart Diagram	**Translation in PROMELA**
className(objID).methodName_thread(v1...vn)	run methodname_className(objID, v1,...vn)

Figure 4.2 Translate method calls to a Thread method

4.2.3.3 The Mapping for Passive Methods

As we explained in section 4.2.1, *passive* methods in our model will be mapped into inline macros.

- Class variables are translated into global variables in PROMELA prefixed by object ids. Whoever uses a class variable should provide the proper object id, as explained in section 4.1.

- Method parameters are not translated into inline parameters directly. This is because the inline parameters have the call-by-reference nature, while here we want to simulate the call-by-value effect, as it is the most commonly used mechanism in object-oriented programming languages. We define input types, output types and related macros to copy the input parameters at the beginning of an inline call and to copy back the output parameters at the end of the inline call.

- Local variables, i.e. variables defined in each method, are also treated specially.

PROMELA inlines allow us to define local variables. However, as we mentioned, the internal structure of an inline construct is *visible to its caller process.* As a consequence,

- if an inline contains local variables, no process can call it twice.
- if two inlines contain local variables with the same name, no process can call both of them

Since these will cause an error of *duplicated definition,* we do not use inline local variables for the method local variables.

Local variables cannot be simply globalized with the inline names either. As a simple example, if two processes call the same inline, they will access the same local variables defined in that inline. In another word, these local variables are no more *local* to each process. Consequently, we need to define local variables *local to each process.*

Our solution is to nest the local variables, input and output parameters of a passive method into its caller. The caller will then pass these variables as the input parameters to the inline. To do so, we make use of the method invocation graph to retrieve the related information.

- Since a method can be called by the same process more than once, if we simply establish the one-to-one relationship between methods and inlines, the translated inline can also be called by the same process more than once. This can be problematic when we have labels in the inlines because in PROMELA, labels cannot be duplicated within the same process. To avoid the confusion on the labels in the PROMELA model, each passive method call action in the statechart diagrams will be translated into a different PROMELA inline, and each label on the statechart diagram may correspond to *a set of* labels in PROMELA model, differentiated by the method calls. More precisely, we make use of the method invocation graph to distinguish the same method in different method calls and translate them into different inlines: these inlines are all the same except that their names are suffixed by the names of different method calls and the labels in each inline are correspondingly translated together with the inline names.

4.2.3.3.1 Record Types for Passive Methods

For each passive method defined in a class diagram, suppose its name is *methodName*, and the class that owns it is named *class*. We define three PROMELA record types:

X_methodName_class_ins   *X_methodName_class_outs*   *X_methodName_class_locals*

Note that the prefix *X_* indicates that they are new record types not corresponding to any class in UML diagrams.

For simplicity, we do not define input/output datatype. An input/output parameter is treated as both an input and an output parameter. The first record type contains the input and input/output parameters of the method and the second contains the output and input/output parameters. The third record type contains some local variables of the method. For each passive method (a passive method as defined in UML class diagrams) *called in the method*, let us denote it as *mCalled* which includes both the method names and the class name, the following three records will be added to X_methodName_class_locals:

```
X_mCalled_ins       x_mCalled_ins;
X_mCalled_outs      x_mCalled_outs;
X_mCalled_locals    x_mCalled_locals;
```

In other words, the input, output parameters and the local parameters of a method will be nested into the local variable of its caller.

Suppose in the class diagram in table 4.1, the method *methodA* in class *A* contains a single integer variable named *localA* and the method does not call any other methods. The method *methodC* in class *C* contains two local variables: an integer named *localC1* and a boolean variable named *localC2*, and the method calls *methodA* in an object of *A*:

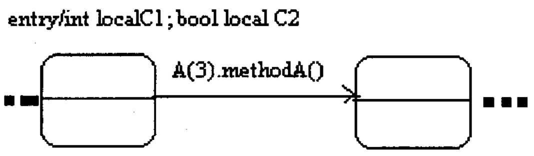


Figure 4.3 A segment of the MethodC statechart diagram

The following PROMELA record types will be generated:

```
typedef X_methodA_A_locals
{  int localA; }

typedef X_methodC_C_ins
{ int cin;
  byte cinout;
}

typedef X_methodC_C_outs
{  bool cout;
   byte cinout;
}

typedef X_methodC_C_locals
{ int localC1;
```

```
    bool localC2;
    X_methodA_A_locals      x_methodA_A_locals;
}
```

Note that during the translation, empty record types, such as *X_A_methodA_ins*, are omitted. For each generated PROMELA record type, an *assign* inline and an *init* inline similar to those in the aforementioned table will also be generated. Their definition will not be repeated here.

Note also that we do not allow recursive calls, thus the type definitions are well-formed.

4.2.3.3.2 Create PROMELA inlines to Represent Passive Methods

We translate a passive method into *a set* of PROMELA inlines according to the *method invocation graph*. This graph can be derived from UML statechart diagrams. As we do not allow recursive calls, it is actually a Directed Acyclic Graph (DAG).

- Each class method will become a node in the graph. The name of the node is the method name suffixed with "_" and the class name.
- If in its statechart diagram, on transition with id *tid*, method *A* calls passive method *B*, a unidirectional arc from node *A* to node *B* will be added to the tree. The arc is labeled by *tid*.

For example, the figure below shows the statechart diagram with three methods, namely the *main* method in class *C*, the passive methods *ma* in class *A* and *mb* in class *B*. The number on the left of a transition shows the id of the transition. The *main* method in class *C* calls method *ma* in the object with id 4 in class *A*, and method *mb* in the object with id 1 in class *B*. The method *ma* in class *A* calls method *mb* in two objects with id 3 and 7 of class B, respectively. The method *mb* in class B does not call any other method.

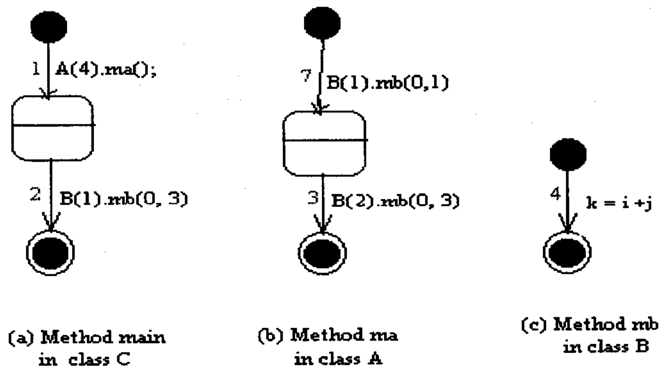


Figure 4.4 Sample Statechart Diagrams

From the sample statechart diagrams above, we can generate the following method invocation graph:

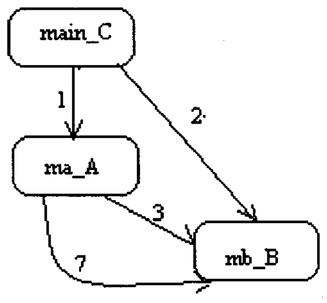


Figure 4.5 Method Invocation Graph for Figure 4.4

An invocation path for node B is a path in the invocation graph from any node A to node B. Given an invocation graph, we can generate the *invocation paths* for each node. Suppose that following the transitions i1 ->i2 ->... ->iN, we can travel from node *A* to node *B*. Let p = i1_i2..._iN. The following inline will be generated for the path:

inline B_p(x_objID, x_in, x_out, x_local) { x_B_clear(x_local); ... }

- The input parameters, *x_in, x_out, x_local* might be omitted if the method does not have any corresponding parameters.
- A passive method may be called multiple times by a method. The inlines related to these calls share local variables. For example, method *mb_B* is called twice in method *ma_A* and the input parameter *x_local* in the inline *mb_B_3* and the inline *mb_B_7* will be the same (See section 4.2.3.3.3.) To erase the changes made by a previous method call, the *x_local* parameter of any passive method inlines will be cleared before being used.

In Figure 4.5, there are four invocation paths for node *mb_B*:

2, 3, 7, 1_3

Accordingly, the following PROMELA inlines will be generated for method *mb* in class *B*:

```
inline mb_B_2(x_objID, x_in, x_out, x_local)      {x_mb_B_clear(x_local); ...}
inline mb_B_3(x_objID, x_in, x_out, x_local)      { x_mb_B_clear(x_local); ...}
inline mb_B_7(x_objID, x_in, x_out, x_local)      { x_mb_B_clear(x_local); ...}
inline mb_B_1_3(x_objID, x_in, x_out, x_local)    { x_mb_B_clear(x_local); ...}
```

The content of these inlines are similar: the only difference is the label name. We suffix all labels in a PROMELA inline with the inline name to avoid confusion.

4.2.3.3.3 The Mapping for Passive Method Calls

In the UML design model, a passive method may be called by an active method or a passive method. Correspondingly, its mapped PROMELA inline may be called in a proctype or in another inline. In either case, the local variable record (x_local) of its caller (see 4.2.3.3.1) will contain an input, an output and a local variable record for the passive method. A passive method call will be mapped as three parts:

1. Assign each field in the input record as the value of the corresponding input parameter.
2. Invoke the proper inline. The name of the inline is constructed from the caller inline/proctype name, the passive method name and the id of the transition on which the method is called. If the caller is an inline, the name of the caller contains information about its own caller, the information of which will then be passed down to the passive methods it called.
3. Update the value of output parameters by assigning them with the values in the output record.

Suppose the passive method *pm* of object *oid* is called on a transition whose id is *tid*. Suppose the transition belongs to the statechart diagram of the method *callerM* in the class *CallerC*. Suppose object *oid* is an instance of class *PC*.

Let us denote the input parameters of *pm* as *in1...inN*, the output parameters of *pm* as *out1...outM*. Suppose on transition *tid*, *iv1,...ivN* are passed as input parameter values, and *op1,...opM* are specified as output parameters.

The method call action on the transition *tid* for the inline *callerM_CallerC_path,* which is an PROMELA inline for method *callerM* in class *CallerC*, will be mapped as:

```
/* x_local is the local variable record of the caller */
atomic
{
  /*Set the input parameters */
  x_local.x_pm_PC_ins.in1 = iv1 ;
   ...
  x_local.x_pm_PC_ins.inN = ivn;
}
pm_PC_path_tid( oid, x_local.x_pm_PC_ins,
                    x_local.x_pm_PC_outs, x_local.x_pm_PC_locals);
atomic
{
   /*Set the output parameter values */
   op1 = x_local.x_pm_PC_outs.out1;
    ...
   opM = x_local.x_pm_PC_outs.outM;
}
```

Note that if method *mMethod* is an active method, *path* does not exist so the inline name will be *pm_PC_tid.*

Our translation guarantees that an inline will never be used twice in a PROMELA proctype. By suffixing all labels in an inline with the name of the inline, conflict on label names is eliminated.

4.4 Translate Statechart Diagrams

In the design model, each method defined in the UML class diagram is assigned a statechart diagram to specify its behavior. In this section, we outline the translation for such statechart diagrams, which will form the internal statements of the PROMELA inlines and proctypes we mapped in the previous section.

In section 4.4.1 we review the statechart diagram in our design model. In section 4.4.2 we introduce the translation for the local signal trigger event. In section 4.4.3 we introduce the translation of variable names, which decides the translation of boolean expressions and variable assignment actions. In section 4.4.4 we introduce how to translate a state and how the translation for each state is organized together to form the translation of a method statechart diagram.

4.4.1 Statechart Diagram Review

As explained in Chapter 3, a statechart diagram in our model contains basic states, composite states and simple transitions. A state can have an entry action. The special entry action is the one in the container state of the statechart diagram, which defines local variables.

A transition may contain a trigger event, a guard condition and an action:

- The trigger event can be a change event or a signal event.
- The guard condition must be a boolean expression.
- An action can be a variable assignment action, a local method call, an IPC action through calling methods on IPC interfaces, the raising of a signal event, or a PROMELA assertion statement.

The states are linked by transitions. From the initial state of the statechart diagram, the control flows from one state to another via an executable transition. The transition is only executable when its source state completes and the executable condition on the trigger event and the guard condition is satisfied. In table 4.6, we listed the possible combinations of the trigger event and guard condition and the executable conditions for each of them:

Transition Type			Executable Condition
Trigger Event		**Guard**	
Signal	**Change**		
No	No	No	Always executable
Yes	No	No	When the signal event occurs
No	Yes	No	Whenever the change event expression becomes true.
No	No	Yes	The guard condition is true at the time the control first exits the source state.
Yes	No	Yes	The guard condition is true when the event occurs. If the guard condition is false when the event occurs, the event is discarded and the control remains on the exiting point of the source state
No	Yes	Yes	The guard condition is true when the change event expression is evaluated to true. If the guard condition is false at that time, the change event is lost and the control remains on the exiting point of the source state.

Figure 4.6 Transition Firing Conditions

4.4.2 Translating Signal Events

A signal event represents the reception of a particular asynchronous signal. In this section, we introduce the mapping of signal events to PROMELA. The introduction is partitioned into four parts. We introduce the PROMELA mapping for the definition, the generation, the dispatching and the reception of signal events.

4.4.2.1 Signal Event Definition

In the design model, signal event types must be defined in UML class diagrams as *Signal* stereotyped classes. Similar to other classes, for a *Signal* class named *S*, if it has one or more class attributes (the signal event is parameterized), we define a PROMELA record type *S_Class* for the class. "Class" is a predefined suffix.

For each signal event type, we define a global channel to hold all generated signal events of that type. Suppose the signal event class type is named *S* and *n* processes uses the signal, the channel is defined as:

```
chan signal_S = [n] of {byte, S_Class};
```

Where the first byte of each element in the channel denotes the process id of the event generator.

4.4.2.2 Signal Event Generation

To simplify the translation, in UML design model, the action for generating a signal event takes the following format:

```
signal(S, v1,...vn);
```

where *signal* is the keyword to distinguish a signal event generation action from a method call action. *S* is the signal event class name, and *v1,...vn* are parameter values. In PROMELA, the event generation action will be translated into the following statement:

```
signal_S!(pid, v1,...vn);
```

where *pid* is the process id of the event generator. A signal event sending action will be modeled as sending a message to the corresponding global signal event channel.

4.4.2.3 Signal Event Dispatching

The effect range of a signal event is local. In other words, only the execution threads belonging to the same process as the one sending the signal event should be able to receive it. If a signal event does not trigger any transition when it is dispatched, the event will be discarded. We do not support reception of signal event in remote methods at this time. Only main methods, thread methods and passive methods called in the main methods and the thread methods can receive signal events.

To realize such functionality, we define a dispatcher process that is responsible for dispatching all signal events. The generation of the dispatcher process takes three steps.

First, the following information is gathered statically from a UML design model:

a. Signal event types.
b. The ids of processes that use a particular signal event type.
c. The maximal listeners allowed in a process for a signal event.

After gathering the information, for each parameterized signal event type and for each process that uses the signal event type, we define a global variable to hold the *current signal event* of the type in that process. Suppose the signal event type is *S* and the process id is *pid*, the global variable will be defined as:

```
S_Class x_signal_S_pid;
```

In addition, we define the following customized datatype at the beginning of the PROMELA program:

```
mtype = {undef, unavailable, available};
```

Suppose in process *pid*, the maximum number of listeners for signal event *S* is *n*. We define an array of size *n* of *mtype* flags for each *S* and *pid*. Each element in the array represents the availability of signal *S* in an execution thread.

```
        mtype x_signal_S_flags_pid[n];
```

For example, the following boolean arrays will be defined for process 1 and process 3 in the previous figure:

```
        mtype x_signal_S_flags_1[4];
        mtype x_signal_S_flags_3[2];
```

The value *undef* means that the flag has not been linked to an active execution thread. The value *unavailable* means the corresponding signal event is not currently available for the thread associated with it. The value *available* means that the event has arrived and is currently available.

With the signal event channels, the related global variables and the related boolean flag arrays in place, we can define the signal event dispatcher process. First, we define the following PROMELA code segment for signal event *S* and process *pid*. Suppose in process *pid*, the maximal number of listener threads for event S is *n (the value max_listener defined in the deployment file for process pid and signal S)*:

```
dispatch(S, pid) =
        signal_S??pid, x_signal_S_pid ->
        if
        ::x_signal_S_flags_pid[0] != undef -> x_signal_S_flags_pid[0] = available;
        ::else -> skip;
        fi;
        ...
        if
        ::x_signal_S_flags_pid[n] !=undef -> x_signal_S_flags_pid[n] = available;
        ::else -> skip;
        fi;
```

In the above code segment, a match receive(i.e. only the event messages with first byte equal to the constant pid will be received) for the is performed on the signal event channel. A signal event generated from process *pid* is retrieved if it exists in the channel and its value saved in the corresponding global variable. If a receptor thread in process *pid* is active, the signal event flag corresponding to it will not be *undef.* Any *S* signal event flag that is not *undef* is set to *available* to indicate that an *S* signal event is currently available for the corresponding thread. The effect range of a signal event is within the process.

Suppose in the system, the signal event types used are *S1…Sn*. For any *Sj* (1<=j<=n), process *pidj1…pidjk* uses them. We define the following *signal event dispatcher* process:

```
active proctype signal_event_dispatcher()
{
  do
  :: atomic
    {
      if
      ::dispatch(S1, pid11);
      ...
      ::dispatch(S1,  pid1k);
      ...
      ::dispatch(Sn, pidnk);
      fi;
    }
   od;
}
```

This process is active from the start. It repeatedly selects a signal event and dispatches it. If no signal event is present in any channels, the dispatcher process is blocked.

4.4.2.4 The reception of a Signal Event

A signal event should be discarded if it does not trigger any transitions in current stage of a statechart diagram. The reception and/or the discarding of signal events are performed by the corresponding execution threads. At the beginning of an execution thread, we reserve a signal event flag for each signal event type used in the execution thread. A signal flag index is saved in a local byte variable of the main/thread method process for later reference.

First, we define the following code segment:

```
flag_reserve(S, pid) =

   byte x_signal_S_flag_index;
   do
   :: x_signal_S_flag_pid[x_signal_S_flag_index] == "undef" ->
      x_signal_S_flag_pid[x_signal_S_flag_index] = "unavailable" ->
      break;
   :: else -> x_signal_S_flag_index = x_signal_S_flag_index+1;
   od;
```

The code segment finds the first undefined flag in the signal event flag array and reserves it for the execution thread by changing the flag to *unavailable.*

After reserving an index for each signal event type received in the thread, we can start translating the reception and the discarding of signal events.

The reception of a signal event is translated in two parts:

1. Save the event parameter values to the corresponding variables.
2. Clear the signal flags.

Suppose in the statechart diagram, a transition includes the following trigger event:

```
signal(S, p1,...pn);
```

If an execution thread in process *pid* includes the transition, The trigger event will be translated as:

```
x_signal_S_flags_pid[x_signal_S_flag_index] == "available" ->
pt1 = x_signal_S_pid.att1; ...ptn = x_signal_S_pid.attn;
```

In the code segment, x_signal_S_flag_index is the flag index saved for the current thread for signal *S*; *pt1...ptn* are the translated variable names and *att1...attn* are the signal event parameter names. The code segment checks if signal S is currently available for the execution thread. If not, the code segment will be blocked. If it is available, the values of the event parameters will be saved into the corresponding variables.

4.4.2.5 Discarding Signal Events

A signal event will be discarded if it does not trigger any transition after dispatched. To achieve that, we add a signal event *clear up* section when the system exits from a none-composite state. A signal event is discarded by setting the corresponding flag to *unavailable.*

Suppose an execution thread in process *pid* is the receptor of signal type S1...Sn. Suppose the flag index of the execution thread for signal type Si is index_i. The following *clean-up* code segment will be added to the beginning of each state of the execution thread:

```
cleanup(pid, S1, index_1, ...Sn) =
signal_S1_flag_pid[index_1] = "unavailable";
...
signal_Sn_flag_pid[index_n] = "unavailable";
```

4.4.3 Attribute Translation

In our model, all boolean expressions are constructed from accessible variables and constants. In addition to being used as guard conditions or change events, boolean expressions are also used in assertion statements. To translate boolean expressions, change event, assignment, etc. we actually only need to consider how to translate the attributes used in them.

The attributes a method can access include its input and output parameters, its local variables and any data attribute of an object within the process boundary.

In statechart diagrams, the data attribute of an object different from the method object is accessed as:

CN(oID).attr

where *attr* denotes the name of the attribute, *oID* denotes the component id of the object (as defined in the deployment file), and *CN* denotes the name of the object class.

In the PROMELA verification model, such an attribute is translated as:

x_CN_getObj(oID).attr

See section 4.1.3 for the definition of the *getObj* macro.

The rest of the attributes include the input, output parameters, the local variables and the data attributes of the method object. All of them are accessed directly (by name only) in the method's statechart diagram, but translated differently in PROMELA verification model. Suppose the attribute name is *p*, and the object class type is *CN*, then:

Attribute Type		**Translation**
Input Parameter	of a Thread Method	P
	of a Passive Metood	x_ins.p
Output Parameter		x_outs.p
Local Variable		x_local.p
Object Data Attribute		CN_getObj(x_objID).p

Figure 4.7 Translate Attributes
(See section 4.2.3.3.1 for the definition of x_ins, x_outs, x_local and x_objID)

4.4.4 Translation of States and Their Outgoing Transitions

Now we have introduced the translation for signal event triggers, signal event generation actions, boolean expressions, assignment actions and assertion actions. The translation for IPC related actions will be introduced in Chapter 5. In this section we introduce how to use these elements to translate states and their outgoing transitions.

First, we translate a basic state as a label. If the state has a name, the label name is:

stateName_methodName_stateID

Where *stateName* is the name of the state. *StateID* is the unique id given to the state in the XMI file. The *methodName* is the translated PROMELA proctype or inline name for the method that the statechart diagram belongs to.

If the state has no name, the label name is:

methodName_stateID

If the state has an entry action, the label is on the beginning of the translated entry action statements. Otherwise, the label is on the beginning of its output transitions.

Suppose the state has *n* output transitions t1...tn and they are partitioned to five sets *Tg, Ts, Tsg, Tc, Tnone:*

- *Tg* includes the transitions that have guard conditions and no trigger events.
- *Tt* includes the transitions that have trigger events and no guard conditions;
- *Ttg* includes the transitions that have both trigger events and guard conditions;
- *Tnone* includes the transitions that have neither guard conditions nor trigger events.

For transiton *j* (1<=j<=n), let us denote the translation for the trigger event of transition *j* as *event_j*, the guard condition as *guard_j*, the action as *action_j*, and the generated label for the target state as *target_j*, the generated label for the source state as *sourceLabel,* we define the following code segment:

```
Tg_transition(j)     =  condition_j ->action_j; goto target_j;
Tt_transition(j) =   event_j -> action_j -> goto target_j;
Tnone_transition(j)  =   1->action_j -> goto target_j;
Ttg_transition(j)    = event_j->
                        if
                       ::guard_j -> action_j ->goto target_j;
                       ::else -> goto sourceLabel;
                       fi;
```

From the above code segments, we define code segment transition(j):

```
    if tj belongs to Tg, transition(j) = Tg_transition(j);
    if tj belongs to Tt , transition(j) = Ts_Tc_transition(j);
    if tj belongs to Tnone, transition(j) = Tnone_transition(j);
    if tj belongs to Ttg, transition(j) = Ttg_transition(j);
```

Using the code segments above, we can define the following PROMELA statements to represent the state and its outgoing transitions. Suppose Tg = {t1,..,tk} (k<=n):

```
sourceLabel: cleanup();
             state_action;
             if
             ::transition(1);
             ...
             ::transition(n);
             else -> /*A guard condition should only be evaluated once. Since PROMELA
                      continuously evaluates all boolean conditions, we must take the transitions
                      with guard condition and no trigger event out after first evaluation. */
               if
               :: transition(k+1);
               ...
               :: transition(n);
               fi;
             fi;
```

After the entry action of the source state is executed, an executable outgoing transition is randomly selected and executed. Note that a transition with both a trigger event and a guard condition may return to the exiting point of the source state. Since the transition belonging to Tg should not be considered if the initial evaluation is false, transitions belonging to Tg are omitted in the *else* statement.

4.4.5 Organize the Translation for the Statechart Diagram

A statechart diagram must directly own one and only one initial state. The statechart diagram must end on an end state.

If more than one transitions are executable when the control leaves the source state, the control will randomly flow from one of them to the next state.

A composite state whose name starts with *atomic* or *d_step* will be translated into a PROMELA *atomic/d_step* block.

Chapter 5 Modeling MOM in PROMELA

In the previous chapter, we defined a verifiable UML model for designing distributed systems. In this chapter, we introduce how the MOM related elements are translated into PROMELA. Such elements include the MOM channels and Publisher, Sender, Receiver and Subscriber interfaces. The translation will be integrated into the PROMELA code the translator generated from the UML model.

The chapter is organized as following: In section 5.1, we will introduce the PROMELA model for point-to-point (PTP) channels. In section 5.2, we will introduce the PROMELA mapping for distributed events.

5.1 PTP Message Passing

In this section, we model PTP message passing mechanism in PROMELA. In section 5.1.1, we will review the features of PTP message passing. As introduced in Chapter 3, in the design model, PTP message passing is realized by declaring PTP channels in the deployment file and creating Sender, Receiver interfaces associated with them in UML diagrams. In section 5.1.2, we introduce the mapping for a PTP channel in PROMELA. In section 5.1.3, we model the Sender interfaces. In section 5.1.4, we introduce the dispatcher processes associated with Sender interfaces. In section 5.1.5, we introduce the mapping for Receiver interfaces.

5.1.1 Features of PTP Message Passing

In Chapter 2, we defined the PTP message passing model for MOM middlewares:

- A PTP channel appears to user as a message queue.
- A PTP channel only accepts messages of a certain datatype. This is due to limitations imposed by PROMELA).
- A message is assigned with a priority: normal or urgent. An urgent message may be retrieved ahead of a normal message, even if the latter is sent earlier.
- A message is assigned with a delivery mode: persistent and transient. A persistent message will never be lost. If the queue is full, the sender will be blocked until the channel becomes available again. A transient message may be lost during a transition: it will be dropped if the queue is full, and it may be lost during a transition. We provide transient delivery model to give users a way to verify system behavior corresponding to the message-lost phenomenon.
- We use Sender interfaces to serialize messages of the same priority, which corresponds to the partial message order enforced by the middleware. Messages from different Sender interfaces are not ordered, which represents the message re-ordering phenomenon that middlewares cannot filter out.
- Message can be retrieved in two ways: it can be read and removed from the channel (receive), or it can be read without been removed (browsed).

5.1.2 The Mapping for PTP Message Channels

In the PROMELA model, a PTP channel defined in the deployment file will be mapped into a PROMELA channel. More precisely, a ptp*Channel* element in the deployment file will be mapped into a PROMELA channel by the following rules:

- The name of the channel is the value of the *name* attribute in the *Channel* element prefixed by *x_ptp_*.
- The size of the channel is the value of the attribute *size* in the *Channel* element.
- The element type of the channel is the message datatype of the PTP channel plus two boolean flags, one for priority and the other for the delivery mode.

5.1.3 The Mapping for Sender Interfaces

In the UML design model, processes send messages to a PTP channel through a Sender interface associated with the channel. A sender interface has three methods, namely *connect*, *send* and *poll*.

In the PROMELA verification model, each Sender interface is mapped to a synchronous gate channel. The name of the channel is the name of the Sender interface prefixed by *x_ptpGate_*.

When a Sender interface is connected to a PTP channel, the gate channel is initialized accordingly and a dispatcher process is activated for the gate channel. The dispatcher process is determined by the PTP channel. The *send* method will send a message to the gate channel, which is then processed by the dispatcher process. The *poll* method will be mapped into a PROMELA statement checking the fullness of the channel:

Sender related Actions in UML	**PROMELA mapping**
Sender s	chan x_ptpGate_s; /*Synchronized gate channel. */
s.connect("tulip")	*x_gateChan_s = [0] of {bool, bool, MT}; /*initialize the gate channel */* run x_ptpDispatcher_tulip(x_gateChan_s); /*starts a dispatcher process */
s.send(urgent, persistent, mt)	x_senderChan_s!1,1,mt;
s.poll();	full(x_senderChan_s); /*returns 1 if the channel is full, 0 otherwise */

Figure 5.2 PROMELA mapping for a Sender interface (tulip is a PTP channel)

In the next section, we explain the functionality of the PTP dispatcher process.

5.1.4 The PTP Dispatcher Process

In the PROMELA verification model, we define a dispatcher proctype for each PTP channel. An instance of a Sender interface connected to the PTP channel will activate a process of that proctype, which manages the messages sent to this channel from a synchronous gate channel representing the Sender interface (See Figure 5.2).

The name of the dispatcher proctype for a PTP channel is the channel name prefixed by *x_ptpDispatcher_*. All dispatcher proctypes for PTP channels are essentially the same: the only difference is their message element type, which is determined by the message element type of the channel.

In the following table, we show the main PTP channel and the dispatcher proctype defined for that channel from a *Channel* element in the PROMELA file:

A "Channel" Element in the Deployment File	**PROMELA MAPPING**

<Channel name = "tulip" buffer_size = 4 channel_type = "ptp" message_datatype = MT">	chan x_ptp_tulip = [4] of {bool, bool, MT};
	proctype x_ptpDispatcher_tulip(chan gate)
	{
	bool priority, persistency;
	MT element;
	x_ptpDispatcher (gate, x_ptp_tulip, priority, persistency, element)
	}

Figure 5.3 PROMELA mapping for ptp channel "tulip"

The major part of any PTP channel dispatcher proctype is the inline "x__ptpDispatcher". The inline "x_ptpDispatcher" repeatedly remove an element from the gate channel, which is a synchronous channel associated with a Sender interface, and send it to the main PTP channel:

```
inline x_ptpDispatcher(gateChan, ptpChan, priority, persistency, element)
{
  xr gateChan; /*declare the process as the sole receiver process for channel "inChan"  to reduce the
                 verification complexity*/
end1: do /*repeat*/
      :: gateChan?priority, persistency, element; /*get a message from inChan */
        if
        :: persistent- > /*Message delivery should be guaranteed */
end2:      mainChan!priority, element;
       :: else ->   /*Message delivery is allowed to fail */
          atomic
end3:     {  if
             :: full(ptpChan) -> skip; /*drop the message if the channel is full */
             :: else ->
                if
                :: ptpChan!priority, element;
                :: skip;   /*randomly drops a message – for testing */
                fi;
            fi;
        }
      fi;
    od;
}
```

Figure 5.4 The x_ptpDispatcher inline

We use the possible interleaving among dispatcher processes between the time a message is retrieved by a dispatcher and the time the message is added to the main PTP channel by the dispatcher to simulate the message re-ordering phenomenon. On the other hand, the messages sent from the same dispatcher process, consequently, from the same Sender interface are ordered, which simulates the partial message-ordering imposed by MOM.

We also model persistent and transient messages. A persistent message (persistency = 1) will never be lost, but may block the sender when the channel is full. A transient message (persistency = 0) will be dropped if the channel is full. In addition, a transient message is randomly dropped, which is useful when user want to examine the error-recovery behavior in message-lost situations. The end labels are added to avoid confusion in deadlock detection – blocking is legal while removing or sending a message.

The following graph deciphers the relationship between a PTP channel *tulip* and its Sender interface A, B and Receiver interface X,Y,Z :

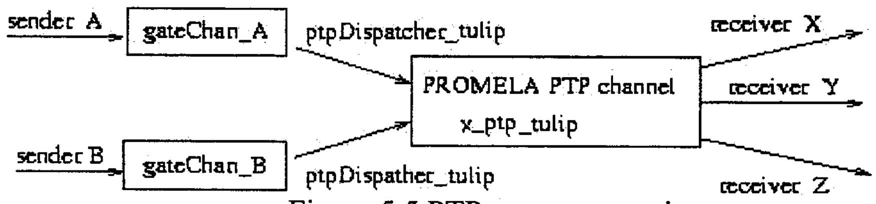


Figure 5.5 PTP message passing

The message priority and the browsing/receiving of the message are modeled on the receiving end – the Receiver interfaces, which will be introduced in the next section.

5.1.5 The PROMELA Mapping for Receiver Interfaces

Receiving a message from a PTP is simple. A receiver interface will be mapped as a reference to the PTP channel to which it connects. A Receiver interface in the UML design model has four methods, namely *connect*, *receive*, *browse* and *poll.* The table below demonstrates the mapping of Receiver interfaces by example:

Receiver Related Actions in UML Statement	PROMELA Code
Receiver rcv	chan x_ptpReceiver_rcv;
Rcv.connect(tulip)	rcv = x_ptpReceiver_tulip;

Mt = rcv.receive();	if
	::x_ptpReceiver_rcv??1,mt; /*retrieve the oldest message*/
	::x_ptpReceiver_rcv?_mt; /* retrieve the oldest urgent message */
	fi

(a,b) = rcv.browse();	if
	::rcv?? <1,a,b>; /*read the oldest message */
	::rcv?_<a,b>; /* read the oldest urgent message*/
	fi

Rcv.poll();	empty(x_ptp_tulip);

Figure 5.6 PROMELA mapping for a Receiver Interface

The translations of method *connect* and method *poll* are straightforward (Compare with the translation for Send objects).

In the mapping for the *browse* and the *receive* method, either the oldest message in the channel is received/browsed, or the oldest urgent message in the channel is received/browsed. In other words, a message with higher priority is given a chance to be retrieved before the normal priority messages ahead of them. The content of the message updates the value of the corresponding variables.

5.2 Distributed Event Channels

In this section, we model distributed event mechanism in PROMELA. In section 5.2.1, we will review the features of distributed events. As introduced in Chapter 3, in the design model, distributed events are realized by declaring distributed event channels in the deployment file and creating/using Publisher, Subscriber interfaces associated with them in UML diagrams. In section 5.2.2, we will introduce the mapping for a distributed event channel. In section 5.2.3, we will model the Publisher interfaces. In section 5.1.4, we will introduce the mapping for Subscriber interfaces.

5.2.1 The Features of Distributed Events

In chapter 2, we defined the distributed event model for MOM middlewares.

- A number of publishers can publish distributed events (sent a message to a distributed event channel). The messages are broadcasted to a number of listeners.
- Events are assigned with priority.
- Events are either persistent or transient: A persistent event must be retrieved by all its subscribers, while a transient event might be lost.

- Events from the same publisher will be serialized, while events from different publishers may interleave.
- A distributed event channel is typed – it can only hold a particular type of messages.

In the UML design model, distributed event channels are defined in the deployment file, Publisher interfaces and Subscriber interfaces are created/used as class attributes and/or method local variables in UML diagrams to use these channels. As we can see, a distributed event is similar to a PTP message in all aspects except that it broadcasts the message to the subscribers.

5.2.2 The PROMELA Mapping for Distributed Event Channels

We map a pcChannel element in the deployment file as a global PROMELA channel (we call it the *main* channel), a boolean *browsed* array and a byte integer *subscriber_count*. The boolean array and the byte integer are used for broadcasting purpose.

A psChannel Element	PROMELA Mapping

<psChannel name = "rose"
buffer_size = "5"
message_datatype = "MT"
max_subscriber = "3">
	chan x_psChan_rose = [1] of {bool, bool, MT};
	byte x_ps__subscriber_count_rose;
	bool x_ps__browsed_rose[3];

Figure 5.7 The PROMELA Mapping for a distributed event channel

The size of the main channel is always *1*. The name of the main channel is the name of the distributed event channel prefixed with "x_psChan". Similar to the PTP channels, the message elements in the channel contains two additional boolean flags, one for priority and the other for persistency.

A main channel receives messages from dispatcher processes, each of such processes associated with a Publisher gate channel. The messages in the main channel are broadcasted to Subscriber listener channels by broadcaster processes. Each of such processes is associated with a listener channel:

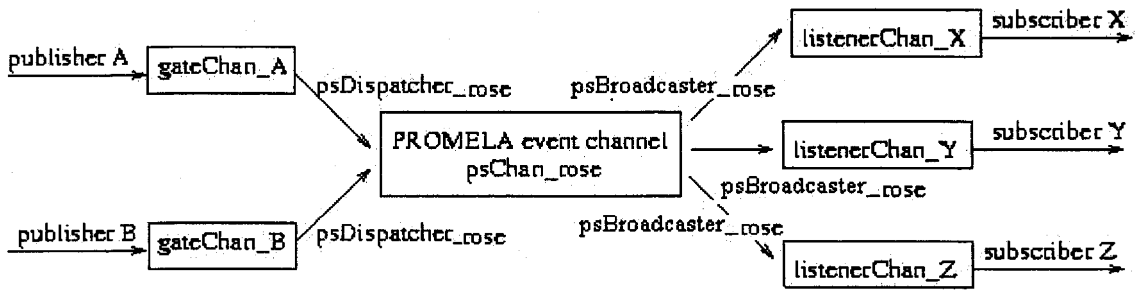


Figure 5.8 The distributed event mechanism

5.2.3 The PROMELA Mapping for Publisher Interfaces

In the PROMELA verification model, A Publisher interface is mapped into a dispatcher process and a gate channel. The dispatcher process is determined by the event channel to which the interface instance connected. Similar to the Sender interfaces, the use of dispatcher processes reserves the event ordering within a Publisher interface while enables message interleaving among different Publisher interfaces.

The main part of a dispatcher process for a Publisher interface is a call to inline x_ps_dispatch:

```
inline x_psDispatch(gateChan, mainChan, element, priority, persistency)
{
  xr pubChan;
 do
 :: inChan?priority, persistency, element;
   gateChan!priority, persistency, element;
 od;
}
```

The definition of dispatcher processes is similar to that of PTP channels and will not be repeated here (see Figure 5.3).

The following table shows the PROMELA mapping for a Publisher interface *pub*:

Publisher Interface	PROMELA Mapping
Publisher pub	chan x_psGate_pub;

pub.connect("rose")	x_psGate_pub = [0] of {bool, bool, MT};
	run x_psDispatcher_rose(x_psGate_pub);

pub.publish(normal, persistent, mt)	x_psGate_pub!0,1, mt;

Figure 5.9 The PROMELA mapping for a Publisher interface

5.2.4 The PROMELA Mapping For Subscriber Interfaces

A Subscriber interface will be mapped into a listener channel, a broadcaster process, and some message receiving statements.

The size of each listener channel is the size of the event channel defined in the deployment file. The name of the listener channel is the name of the Subscriber interface prefixed by *x_psListener_*.

The broadcaster process is determined by the distributed event channel to which the Subscriber interface instance connects. The process will repeatedly browse the corresponding main channel and send the browsed element to the listener channel. To avoid browsing the same event twice, the dispatcher process uses an element in the boolean *browsed* array. The message is deleted from the main channel when all its dispatcher processes have browsed it. The integer *subscriber_count* keeps the number of Subscriber interface instances connected to the main channel. The detail of the broadcaster processes will be introduced in the next section.

For example, a Subscriber interface *sub* connected to the distributed event channel *rose* (see Figure 5.7), the PROMELA mapping for *sub* will be:

Subscriber Interface	**PROMELA Mapping**
Subscriber sub	chan x_psListener_sub;
Sub.connect("rose");	x_psListener_sub = [5] of {bool, MT}; run x_psBroadcaster_rose(x_psListener_sub);
mt = sub.consume();	if ::x_psListener_sub??1,mt; ::x_psListener_sub?_mt; fi
Sub.empty()	atomic {/* drop all elements in the listener channel */ do ::empty(x_psListener_sub) -> break; ::else -> x_psListener_sub ? _,_,_; od; }
Sub.poll()	empty(x_psListener_sub)

Figure 5.10 The PROMELA mapping for a Subscriber interface

5.2.5 The Broadcaster Processes

A broadcaster process is associated with a distributed event channel and a listener channel. For each *psChannel* element in the deployment file, which defines a distributed event channel, we create a broadcaster proctype. Each Subscriber interface instance connected to the event channel will activate a broadcaster process from the proctype, using the listener channel as the input parameter (see Figure 5.10).

The functionality of a broadcaster process is to repeatedly browse the main channel and send the browsed element to the listener channel. A broadcaster process will never browse the same element twice. To achieve this, broadcaster processes take turns to reserve a *browsed* flag in the *browsed* array. The array index is determined by the number of existing broadcasters (*subscriber_count*) of that distributed event channel. The flag is set to 1 before browsing. After browsing an element, the broadcaster process set its *browsed* flag to *0*. The flag will not be set back to *1* until the browsed element is deleted from the main channel.

An element will not be deleted from the main channel until it has been browsed by all the broadcasters, in other words, when no flag in the *browsed* array is of value *1*. After the element is deleted, all element in the *browsed* array that is reserved by a broadcaster process (index<subscriber_count) will be refreshed to *1* so the broadcaster processes can start browsing the next element.

An element is sent to the listener channel in the same way as a PTP message.

Broadcaster protocols are essentially the same, except that they refer to different main channels, *subscriber_count* integers and *browsed* arrays. The element type may also defer from one another. Here we explain the broadcaster proctypes through an example.

```
inline x_psBroadcaster_rose( lsnChan,)
{
  /*Reserve a boolean "browsed" flag, update "subscriber_count" */
  atomic
  {   byte index; /*"browsed" flag index*/
      xs lsnChan; /*Reduce verification complexity, set the process as the sole sender for lsnChan*/
      index = x_ps_subscriber_count_rose;
      x_ps_subscriber_count_rose = x_ps_subscriber_count_rose+1; /*update subscriber_count*/
      x_ps_browsed_rose[index] = 1;
  }

  MT element; /*event element */
  bool persistency, priority;  /*persistency = 1: persistent event.   priority = 1: urgent event
                                 persistency = 0: transient event.   priority = 0: normal event*/
  byte count; /* A counter variable for setting reserved "browsed" flag back to 1*/
  do
  :: atomic
    {  x_psChan_rose?[_,_,element]; /*an element is available?*/
       x_ps_browsed_rose[index] > 0  -> /*if the element hasn't been browsed*/
       x_ps_browsed_rose[index] = 0;    /* browsing*/

       if
       ::/*If not everyone has finished browsing, browse the element*/
         x_ps_browsed_rose[0] | x_ps_browsed_rose[1] | x_ps_browsed_rose[2]
         -> x_psChan_rose ?<priority, persistency, element>; /*browse*/
       :: /*Otherwise, retrieve the element, refresh "browsed" flags*/
          else -> x_psChan_rose?priority, persistency, element;  /*read & delete*/
                do /*update flags*/
                :: count<x_ps_subscriber_count_rose ->
                   x_ps_browsed_rose[count] = 1;
                   count = count+1;
                ::  else -> count = 0; break;
                od;
        fi;

        /*send the browsed element to the listener channel according to persistency value,
          compare to Figure 5.4*/
        if
       ::persistency  -> lsnChan!priority, element;
       else -> if
               ::full(lsnChan) -> skip;
               ::else -> if
                         ::lsnChan!priority, element;
                          ::skip;
```

```
                    fi;
                fi;
            fi;
        }
      od;
    }
```

As shown in the example, we simulate the broadcasting of distributed events through browsing. We also simulate the event-reordering phenomenon among publishers. To reduce complexity, however, we did not model event-reordering phenomenon among subscribers. In our model, events will arrive in all subscribers in the same order. In other words, we modeled a *centralized* distributed event system, in which events are processed by a unique dispatcher process.

Chapter 6 The Remote Object System

A CORBA remote object system includes one or more ORBs, the object implementations and the clients. A client issues requests. The ORB is responsible for finding the object implementation for the request, to prepare the object implementation to receive the request, and to communicate the data that make up the request. The object implementations, managed by object adapters, are responsible for servicing the requests.

In Chapter 3, we introduced the CORBA-style remote object system in the design model. In this chapter, we will introduce the corresponding PROMELA implementation.

The basic idea is to define the request channels for clients to transfer requests and the result channels to give back results to the clients. Different worker-thread processes are generated and linked to different request channels according to POA thread policies and ORB thread policies, they represent the object implementations.

We will introduce the mapping in three parts. First, we introduce the mapping for Stub instances, which represent the client interfaces. The finding of object implementation is also included in this section. Secondly, we introduce the generation of worker threads, according to ORB thread policy and POA thread policies. Thirdly, we compare our work with others.

6.1 The Mapping for the Clients -- Stub Instances and Response Objects

A Stub instance in the UML design model represents a client. Each stub will be translated into a set of channels. The channels will be defined at the time the Stub instance is created. When a Stub instance is created, it is specified as a Stub instance for a remote object type. For each method defined on the object interface, two channels are created, one for sending request and one for getting result. Suppose the Stub instance is named *s* and the method on the remote object interface is named *m*. The channel will be defined as:

```
chan stub_s_m;
```

The channels will be initialized when the Stub instance is bound to a remote object. The initialization of the request channels depends on the thread policy of the POA that manages the remote object. A new channel may be created and then passed into a worker thread, or the channel may be a reference to an existing channel (shared with other clients). Suppose the Stub instance is bound to a remote object with id *oid*, which has n method named *m1…mn*.

Suppose the IDL class name for object *oid* is *C*. The following table shows the translation of a Stub instance bound to *oid*. (1<=j<=n)

POA Thread Policy	Binding Translation
main_thread	… stub_s_mj = mj_oid_in; …
thread_per_POA	… stub_s_mj = mj_oid_in; …
thread_per_obj	… stub_s_mj = mj_oid_in; …
thread_pool(n)	… stub_s_mj = mj_oid_in; …
Thread_Per_Client	… stub_s_mj = [0] of {byte, X_mj_C_ins, chan); … run worker_thread_C(stub_s_m1, …, stub_s_mn);

Figure 6.1 Translation for Binding on the server side

The channel mj_oid_in is for method mj in remote object oid, which is shared by the clients to issue mj requests to oid. Unless the POA who manages the remote object uses the *Thread_Per_Client* thread policy, mj_oid_in will be defined globally as:

chan mj_oid_in = [0] of {byte, X_mj_C_ins, chan};

The *byte* flag indicates the style of the method call (synchronized, asynchronized, deffered) and the element of the datatype X_mj_C_ins contains the input parameters of the method, if there is any. The channel element is the channel for sending back result.

The definition of worker_thread_C will be introduced later. For each Stub instance, a return channel is defined for each method. For method mj, the return channel will be defined as:

chan stub_s_mj_return = [0] of {bool, X_mj_C_outs};

The boolean flag is added for those methods without returning values so they can be called synchronously or asynchronously.

After binding, the Stub instance can be used to send requests to the remote object *oid*. The request can be sent synchronously, asynchronously or deferred synchronously, using method send_sync, send_oneway and send_deffered, accordingly. Suppose mj has n parameters, p1, …pn. We denote the input parameters as pin_1 … pin_k and the output parameters as pout_r,…pout_n (r<=k+1). The translation of different calling style is given in the next table:

Method Call Mode	Translation
send_sync(mj, p1,…pn);	stub_s_mj!0, pin_1,…pin_k, stub_s_mj_return; stub_s_mj_return?_, pout_r,…pout_n;
send_oneway (mj, p1,…pn);	stub_s_mj!1, pin_1,…pin_k;
send_deferred (mj, p1,…pn, rsp)	stub_s_mj!2, pin_1,…pin_k , stub_s_rsp_mj rsp_return? , *pout_r*,…pout_n;

Figure 6.2 Translation for method calls of different synchronization model

In the *send_deferred* mode, a response object is used as an input parameter. A response object is mapped to a special return channel. Unlike the return channels associated with a method of a Stub instance, the channel of a response object is an asynchronous channel, which hold the result for the client to poll later. In our model, a response object is associated with a particular remote method, and no more than two remote method calls should be pending on a response object at any time.

Response Object	**Translation**
Response rsp	chan rsp_return;
rsp = new Response(C, mj)	rsp_mj_C_return = [2] of {bool, C_mj_outs};
rsp.get_response()	rsp_return?_pout_r,...pout_n;
rsp.poll_response()	empty(rsp_return)

Figure 6.3 Translation for the Response objects

For simplicity, we did not show the mapping of the Response object name in the above table as in section 4.1

In this section, we have introduced the translation of clients, including how to send requests and how to get results. In the next section, we introduce how to decide the binding between a Stub instance and an object implementation.

6.2 Binding an Object Implementation to A Stub Instance

Binding an object implementation to a Stub instance, in the PROMELA mapping, is equal to determine the id of the remote object and initialize the set of stub channels accordingly (See Section 6.1).

Three binding options were given in the design model: A Stub instance may be bound randomly, bound to a remote object with particular id, or bound to a remote object within a particular POA.

During the binding, all remote objects are of the same priority. In other words, if two remote objects satisfy the binding option, they will have the equal change to be bound to the Stub instance, despite the object location, the available thread resources, etc.

Suppose *stub* is created as a Stub instance for IDL interface type *C*. Suppose in the system, *C* type remote objects is scattered in POAs named pid_1...pid_k. Suppose in pid_j, 1<=j<=k, remote objects oid_1j...oid_mj are *C* type remote objects. (1j<=mj). Suppose C has n methods, denoted as m_1...m_n. Suppose that the ORB that pid_j connects to be orbIDj.

Let us define:

```
set_stub_channels_obj(stub, oid, orbID, 0 ) =
atomic{
stub_s_m_1 = m_1_oid_in;
...
stub_s_m_n = m_n_oid_in;
}
```

Denoted by the last parameter ("0"), the above code segment is for binding the Stub instance to an object that resides in a POA with thread policy other than *thread_per_client*. The ORB that the POA connects to is *orbID*. The channels are set to reference to the existing request channels belonging to the remote object.

```
set_stub_channels_obj(stub, oid, orbID,1 ) =
atomic{
stub_s_m_1 = [0] of {byte, X_m_1_C_ins, chan);
...
stub_s_m_1 = [0] of {byte, X_m_n_C_ins, chan);
}
run worker_thread_C( oid, orbID, stub_s_m1, ..., stub_s_mn);
```

Denoted by the last parameter ("1"), the above code segment is for binding the Stub instance to an object that resides in a POA with thread policy being *thread_per_client*. The ORB that the POA connects to is *orbID*. The request channels are specifically created for this Stub instance (client), and a worker thread is activated to perform the implementation tasks of the remote object (server).

```
set_stub_channels_poa(stub, p, orbID) =
atomic{
if
  :: set_stub_channels_obj(stub, obj_1p, orbID, (poa_thread_policy ==
thread_per_client));
  ...
  :: set_stub_channels_obj(stub, obj_mp, orbID, (poa_thread_policy ==
thread_per_client));
fi;
}
```

The above code segment randomly binds the stubs object to a suitable remote object in POA *p* that is connected to ORB *orbID*.

```
set_stub_channels_rand(stub) =
atomic{
  if
  ::set_stub_channels_poa(stub, pid1, orbID1);
     ...
  ::set_stub_channels_poa(stub, pidk, orbIDk);
  fi;
}
```

The above code segment randomly binds the Stub instance to a suitable remote object in the system.

The binding will be translated as:

Binding Action	Translation
stub.bind(OBJ_ID, o)	set_stub_channels_obj(stub, o, (poa_thread_policy == thread_per_client));
stub.bind(POA_ID, p)	set_stub_channels_poa(stub, p);
stub.bind()	set_stub_channels_rand(stub);

Figure 6.4 Translation for binding on client side

After binding, the communication between the client (the Stub instance) and the server (object implementation) does not need monitoring. Clients send request as message to the channel, and the server will retrieve the message from the channel.

6.3 Object Implementation – The Definition of Worker Threads

In this section, we introduce the worker threads. Remote objects are managed by POAs. According to the thread policy, a POA is given some thread resources. A remote method call can only be serviced if there is thread resource available. Otherwise the call will be blocked until so.

In Chapter 3, we have introduced the translation for local objects. The translation for remote objects is similar. All methods on the IDL interface will be translated as passive methods. The variable set/get methods are translated accordingly. In this section, we define worker thread processes, which retrieves remote method call requests from channels and invoke remote method inlines accordingly. The definition and the creation of worker threads are determined by the ORB thread policy and the POA thread policy. Worker thread processes have a similar structure, the only difference being their scope.

6.3.1 ORB Thread Policy

In our PROMELA model for CORBA distributed object system, for simplicity the clients (Stub instances) and the object implementation (worker threads) are linked together directly by request channels. In reality, such requests are first sent to the ORB itself, then the ORB dispatch the requests to the corresponding POAs. If the ORB is single_thread, the transfer of the request by ORB may block the execution. If the ORB is multi_thread(we suppose the thread resource is unlimited for an ORB), the blocking will not occur. We suppose the communication between ORBs will never block.

Suppose in the deployment file, n ORBs are defined, with id 1...n. If ORB i...j are single_thread ORB and the rest are multi_thread ORB, following code segment will be added to the PROMELA file:

```
mtype orb_busy[n]; /*mtype is defined in Chapter 5. The value can be undef, unavailable
                        or available*/
...
init
{
  /* Set the ORB flag */
  orb_busy[i] = available;
  ...
  orb busy[j] = available;
  ...
}
```

6.3.2 The Main_Thread Policy

An ORB owns a special execution thread, which is referred to by the POAs as the *main thread.* If the thread policy of a POA is specified as *Main_Thread*, any service requests to the remote objects managed by the POA will be serviced by the main thread. Since all POAs with the *Main_Thread* thread policy will share the thread, the concurrency is cut to minimum and the concurrency control task will be greatly simplified or even eliminated. The drawback is that such a thread policy can easily cause deadlock due to the competition of the main thread.

For each ORB defined in the deployment file, if any of the POAs under it specified its thread policy as *Main_Thread*, we define a worker thread to handle all the remote methods in such POAs.

Suppose the ORB id is *orbId.* Suppose ORB orb_id has n POAs with *Main_Thread* thread policy. Suppose the POA ids are poa_1...poa_n. Suppose poa_j (1<=j<=n) manages *jm* number of remote objects, with object ids obj_j1...obj_jm. Suppose the remote object obj_jk(j1<=jk<=jm) has jkz methods, the name of these methods are m_jk1...m_jkz.

Suppose the IDL interface type for object obj_jk is Cobj_jk. Let us define the following code segment:

```
create_channels_obj(o) =
        chan m_o1_o_in = [0] of {byte, X_m_o1_Co_ins, chan);
        ...
        chan m_oz_o_in = [0] of {byte, X_m_oz_Co_ins, chan);
```

```
create_channels_poa(p) =
        create_channels(obj_p1);
        ...
        create_channels(obj_pm);

define_element_obj(o) =
        X_m_o1_Co_ins x_m_o1_Co_ins;
        X_m_o1_Co_outs x_m_o1_Co_outs;


        ...
        X_m_oz_Co_ins x_m_oz_Co_ins;
        X_m_oz_Co_outs x_m_oz_Co_outs;

define_element_poa(p) =
        define_element_obj(obj_p1);
        ...
        define_element_obj(obj_pm);


process_request(m,o, C, orbID) =
/*get a request for method m in remote object o. The IDL type for 0 is C */
   m_o_in?style, x_m_C_ins, returnChan; /* style is a byte flag defined beforehand */
   atomic
   {
   if
   :: /* wait for the flag if the orb is single thread orb*/
     orb_busy[orbID] == available ->
     /* reserve flag*/
     orb_busy[orbID] == unavailable;
   :: /* continue if the orb is multi_thread orb*/
     orb_busy[orbID] == undef -> skip;
   fi;
   }
   m_C(o, x_m_C_ins, x_m_C_outs);
   if
   :: style == 1 -> skip; /*asynchronized call does not require return. */
   :: else ->returnChan!1, x_m_C_outs;
   fi;
   /* release flag if the orb is a single thread orb */
   atomic
   {
    if
    :: orb_busy[orbID] == unavailable -> orb_busy[orbID] = available;
    :: else -> skip;
    fi;
```

```
  }
```

In PROMELA, the ORB and the *Main_Thread* POAs under it is translated as:

```
/*create global request channels */
create_channels_poa(p1)
...
create_channels_poa(pn)

active proctype worker_thread_orb_orbID()
{
   /* mark the worker thread process as the sole receiver of the global request channels
   above to cut down verification complexity*/
    xr m_111_obj_11_in;
    ...
    xr m_nmz_obj_nm_in;

   /* define elements to hold input parameters and output channels for each remote
      method    */
  define_element_poa(p1)
  ...
  define_element_poa(pn)

 /* define a return channel reference and a flag for the call's synchronization style*/
   chan returnChan;
   byte style;

  /* repeatly remove a request from a channel and process it */
  do
  ::process_request(m_111, o_11, Co_11);
  ...
  ::process_request(m_nmz,o_nm, Co_nm);
  od;
}
```

The scope of the worker thread process is the object implementations in any remote objects managed by a *Main_Thread* POA under ORB *orbID.*

6.3.3 The Thread_Per_POA Policy and the Thread_Pool Policy

A POA may also be assigned with its own threads to handle all remote objects in it. If the thread policy is *Thread_Per_POA*, one thread is used to handle all remote objects in the POA. If the thread policy is *Thread_Pool(psize)*, 1<=psize<=9, a set of threads are used to handle them. The *Thread_Per_POA* policy is actually the same as the *Thread_Pool(1)* policy.

Suppose the thread policy for a POA with id *pid* is *Thread_Per_POA*. Suppose the POA manages m objects obj_1...obj_m. Suppose the remote object obj_k (1<=k<=n) has kz methods, named m_k1...m_kz, accordingly. Suppose the ORB the POA connects to is *orbID*. The POA and the remote objects managed by it is translated as:

```
create_channels_poa(pid)
active proctype worker_thread_poa_pid()
{
    xr m_11_obj_1;
    ...
    xr m_mz_obj_m;
   define_element_poa(pid)

   chan returnChan;
   byte style;
  do
  ::process_request(m_11, o_1, Co_1, orbID);
  ...
  ::process_request(m_mz,o_m, Co_m, orbID);
  od;
}
```

Suppose the thread policy is *Thread_Pool(psize)*, we only need to change the second line above to:

```
active [psize] proctype worker_thread_poa_pid()
```

It activates *psize* processes from the proctype instead of one. The scope of the worker thread is the remote objects within the POA.

6.3.4 The Thread_Per_Object Policy and the Thread_Per_Client Policy

For a POA with the *Thread_Per_Obj* policy or *Thread_Per_Client* policy, we need to define worker thread whose management scope is a particular remote object. The only difference between the two thread policy is that for a POA with *Thread_Per_Object* policy, a worker thread is activated from start for each object. For a POA with *Thread_Per_Client* policy, a worker thread will be activated by a client (a Stub instance) of a remote object.

For each object with id *oid* in the POA, suppose the methods within it are named m1...mz and the IDL class for oid is *C*. Suppose the ORB the POA connects to is *orb*. First, we define the following code segment:

```
process_request_chan(channel, m, o, C, orbID) =
/*get a request for method m in remote object o. The IDL type for o is C */
   if
   :: orb_busy[orbID] == available -> orb_busy[orbID] == unavailable;
   :: orb_busy[orbID] == undef -> skip;
   fi;

   channel?style, x_m_C_ins, returnChan; /* style is a byte flag defined beforehand */
   m_C(o, x_m_C_ins, x_m_C_outs);
   if
   :: style == 1 -> skip; /*asynchronized call does not require return. */
   :: else ->returnChan!1, x_m_C_outs;
   fi;
   if
   :: orb_busy[orbID] == unavailable -> orb_busy[orbID] == available;
   :: else -> skip;
   fi;
```

The code segment is similar to code segment process_request(m, o, C). The only difference is that the request channel for method *m* in remote object *o* is no longer fixed as the global channel, but as an input parameter.

If the POA thread policy is *Thread_Per_Client*, the following translation will be added to the PROMELA file:

```
 proctype worker_thread_C(byte oid, byte orbID, chan c1, ...chan cz)
{
   xr c1;
   ...
   xr cz;
  define_element_obj(oid)

  chan returnChan;
  byte style;
 do
 ::process_request_chan(c1, oid, C, orbID);
 ...
 ::process_request(cz, mz,oid, C, orbID);
 od;
}
```

If the thread policy is *Thread_Per_Client*, a process will be activated from the proctype whenever a Stub instance is bound to the object. (See Section 6.1)

If the thread policy is *Thread_Per_Object*, a process will be created from the proctype for the object in the init process. In addition, a set of global request channels are added for the object:

```
create_channels(oid)
init
{ ...
  run worker_thread_C(oid, orb, m1_oid_in, ...mz_oid_in);
  ...
}
```

6.4 Conclusions

In this chapter, we defined our PROMELA mapping for CORBA distributed object system. The translation can be easily adapted to other distributed object systems, such as Java RMI or DCOM.

Our translation is inspired by the research work in [5, 10, 11]. Compared to other research work, our translation focuses on the distributed system built on top of the distributed object system, rather than the distributed object system middleware itself.

The translation is automatic. Instead of trying to represent the structure of ORB and POAs, the PROMELA code we give represents their behavior. The rigid proof of the correctness of the translation is beyond the scope of this thesis.

We discuss both the single thread ORBs and the multi_thread ORBs. We modeled all common thread policy for object adapters, including main_thread, thread_per_poa, thread_pool, thread_per_object and thread_per_client. The remote method calls can have parameters. The part we left out intentionally is the Inter-ORB communication: we assume such communication is always successful and never blocks. Readers interested in this part can refer to the work done in [10].

Chapter 7 Related Work & Conclusion

The research work presented in this thesis can be partitioned roughly into three parts:

- Translate UML diagrams to PROMELA.
- Define MOM client interface in UML and model MOM behavior in PROMELA.
- Model DOS middleware behavior in PROMELA.

The three parts are integrated together to form a verification tool for distributed systems built on top of middlewares.

Our UML diagram translation is closer to the research work done on translating programming language to PROMELA [2 3 6] than other UML-PROMELA translations [12 13 15 18]. The reason is that the latter focus on the de-composition of UML diagram structures, such as the de-composition of composite states with concurrent regions, the complex transitions, etc. and the rigid correctness proof for such de-composition. The execution elements in such research are cut to minimal. For example, in [12] the authors only consider transitions with non-parameterized signal events. Execution elements such as guard conditions, call events, change events are all omitted, which severely limits the expression power of UML statechart diagram. In our approach, we only consider UML statechart diagrams with very simple structure – each statechart diagram represents a class operation. On the other hand, we distinguish change events from guard conditions, which some researchers have ignored [18]. The signal events in our model can have parameters. We have modeled method calls, including remote ones. We have modeled distributed PTP message passing and distributed events through specially defined middleware client interfaces. Some verification information is also included in our model, such as assertion statements, legal end state, progress state, etc. which allows a novice user to maximally benefit from the power of SPIN. In addition, we do not limit the users on the deployment: Users can create as many objects from a class as needed, and the deployment can be changed easily on the deployment file without changing UML diagrams. In some other approaches ([18] for example), users are allowed to have only one object per class, and one statechart diagram per class.

We integrated the middleware services into the PROMELA verification model generated from the UML model. Message oriented middlewares have not been considered by others. We realized the PTP message passing and the distributed event broadcasting mechanism in PROMELA. DOS middlewares, such as CORBA, have been examined by others [10,11]. Our approach, however, focused on the correctness of the distributed system built on top of such middlewares, instead of verifying the correctness of the middlewares themselves. In addition, our translation is automated.

While formal techniques have long been recognized as invaluable for improving the quality of design specifications, they are also mocked as "toys for mathematicians". They can only handle small applications and they are hard to understand and to use, unless by someone with strong mathematics and logic background.

Numerous research works have been done to improve the acceptance of formal techniques. Some of them aimed at increasing the power of formal techniques. Nowadays model checking tool such as SPIN can handle over 200,000 states. Others aim at the composition and the de-composition of design specifications, so that the formal techniques can be used to verify a design specification block-by-block, layer-by-layer. Still, there are research work aiming at simplifying the application of formal techniques so that they are acceptable by common programmers. Our research belongs to the third approach. We simplify the verification of distributed systems with middleware by providing proper UML elements and the automated translation to PROMELA. SPIN may still be considered a "toy" by others, but at least we can make it a toy for everyone.

References

[1] A. Campbell, G. Coulson, and M. Kounavis. "Managing Complexity: Middleware Explained." IT Professional, IEEE Computer Society, 1:5, September/October 1999, 22 - 28

[2] T. Cattel. Modeling and verication of sC++ applications. In Proc. of the Tools and Algorithms for the Construction and Analysis of Systems, LNCS 1384, pages 232-248. Springer-Verlag, 1998.

[3] J. Chen. On verifying distributed multithreaded Java programs. Software Quality Journal, 8:321-341, 1999.

[4] R. Cleaveland and S. Sims. The NCSU concurrency workbench. In Computer-Aided Verification (CAV'96), LNCS 1102, pages 394-397. Springer-Verlag, 1996.

[5] G. Duval. Specication and verication of an object request broker. In Proc. of the 20th International Conference on Software Engineering, pages 43-52, 1998.

[6] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. International Journal on Software Tools for Technology Transfer, 2(4), April 2000.

[7] C. A. R. Hoare. Communicating Sequential Processes. Prentice Hall Int., London, 1985.

[8] G. Holzmann. The Design and Validation of Computer Protocols. Prentice Hall, 1991.

[9] G. Holzmann. The model checker SPIN. IEEE Transactions on Software Engineering, 23(5), May 1997.

[10] M. Kamel and S. Leue. Validation of remote object invocation and object migration in CORBA GIOP using Promela/Spin. In Proceedings of SPIN Workshop'98, Paris, France, November 1998.

[11] N. Kaveh and W. Emmerich. Deadlock detection in distributed object systems. In Proc. of the Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9), pages 44-51. ACM Press, 2001.

[12] D. Latella, I. Majzik, and M. Massink. Automatic verication of a behavioural subset of UML statechart diagrams using the SPIN model-checker. Formal Aspects of Computing, 11:637-664, 1999.

[13] J. Lilius and I.P. Paltor. Formalizing UML state machines for model checking. In Proc. of the 2nd International Conference on United Modeling Language (UML'99), LNCS 1723, pages 430-445. Springer-Verlag, 1999.

[14] K. L. McMillan. Symbolic Model Checking. Kluwer Academic Publishers, 1993. 14

[15] E. Mikk, Y. Lakhnech, M. Siegel, and G.J. Holzmann. Implementing statecharts in PROMELA/SPIN. In Proc. of the 2nd IEEE Workshop on Industrial Strength Formal Specication Techniques, pages 90-101, 1998.

[16] Object Management Group, Common Object Request Broker Architecture, Version 3.02, (Dec. 2002) http://www.omg.org/cgi-bin/doc?formal/02-12-02.

[17] Object Management Group, *Unified Modeling Language Specification, Version 1.5*, (March 2003), http://www.omg.org/cgi-bin/doc?formal/03-03-01.

[18] T. Schafer, A. Knapp, and S. Merz. Model checking UML state machines and collaborations. Electronic Notes in Theoretical Computer Science, 47:1-13, 2001.

[19] Sun Microsystems Corp., Java Message Service Specification, version 1.02b, August 27, 2001

[20] T. Y. C. Woo and S. S. Lam. Authentication for distributed systems. Computer, 25(1):39{52, January 1992

[21] M. Yong and M. Butler. Tool support for visualizing CSP in UML. In Proc. of the 4th International Conference on Formal Engineering Methods (ICFEM'02), pages 287-298. Springer Verlag, 2002.

[22] World Wide Web Consortium (W3C). Extensible Markup Language (XML) 1.0, second edition, October 2000. http://www.w3.org/TR/RECxml

Vita Auctoris

Hanmei Cui was born in 1973 in Xi'an, China. She graduated from the No. 1 High School of Xi'an in 1991. From there she went on to the University of Zhejiang where she obtained a B.E. in Information & Electronics Engineering in 1995. She is currently a candidate for the Masters Degree in Computer Science at the University of Windsor and hopes to graduate in Winter 2003.