On Customizing the UML for Modeling Performance-Oriented Applications *

Sabri Pllana and Thomas Fahringer

Institute for Software Science, University of Vienna,
Liechtensteinstraße 22, 1090 Vienna, Austria
{pllana, tf}@par.univie.ac.at

Abstract. Modeling of parallel and distributed applications was a preoccupation of numerous research groups in the past. The increasing importance of applications that mix shared memory parallelism with message passing has complicated the modeling effort. Despite the fact that UML represents the de-facto standard modeling language, little work has been done to investigate whether UML can be employed to model performance-oriented parallel and distributed applications. This paper provides a critical look at the utility of UML to model shared memory and message passing applications by employing the UML extension mechanisms. The basic idea is to develop UML building blocks for the most important sequential, shared memory, and message passing constructs. These building blocks can be enriched with additional information, for instance, performance and control flow data. Subsequently, building blocks are combined to represent basically arbitrary complex applications. We will further describe how to model the mapping of applications onto process topologies.

1 Introduction

Effective performance-oriented program development requires the programmer to understand the intricate details of the programming model, the parallel/distributed architecture, and the mapping of applications onto architectures. Numerous approaches [5, 1, 2] have been introduced to graphically model applications, and their mapping onto parallel and distributed architectures. These approaches, however, have only been moderately successful due to modeling languages that have not been based on standards, restrictions to specific program paradigms (e.g. message passing only), and limited modeling capabilities.

With the appearance of the Unified Modeling Language (UML) [7], however, things have changed. UML is the de-facto standard visual modeling language which is a general purpose, broadly applicable, tool supported, industry standardized modeling language. Although UML has been extensively studied and tested in the domain of distributed computing, in particular for client server and web applications, hardly any effort has been undertaken to examine its practicability and effectiveness for performance-oriented scientific applications that

* The work described in this paper is supported by the Austrian Science Fund as part of Aurora Project under contract SFBF1104.

exploit message passing, shared memory, data parallelism, and mixed forms of parallelism (e.g. shared memory combined with message passing). Even though the majority of scientific applications have been developed based on procedural languages (Fortran and C), UML is a promising alternative to model non-object oriented parallel and distributed applications as well.

In this paper, we provide a critical look at the utility of UML to model shared memory and message passing applications. We describe how to use the UML extension mechanisms to customize UML for the domain of parallel and distributed computing by using a subset of UML, namely class, activity, and collaboration diagrams. Class diagrams are employed to build the structural model of distributed and parallel architectures; activity diagrams for representing computational, communication, and synchronization operations; collaboration diagrams for describing process topologies and the mapping of applications to process topologies. We describe a set of UML building blocks that model some of the most important constructs of message passing and shared memory parallel paradigms. The building blocks have been largely motivated by the Open Multi Processing (OpenMP) [3] and the Message Passing Interface (MPI) [12] standards. Based on the pre-defined UML building blocks we can develop models for large and complex applications. We will further describe how to model the mapping of distributed/parallel applications to process topologies. Based on our work we observed that the core UML does not sufficiently support modeling of performance-oriented parallel and distributed applications. However, the UML extension mechanisms in particular stereotypes appear to be very useful to customize UML for the domain of performance-oriented distributed and parallel computing. Furthermore, we believe that the definition of a UML profile for the domain of performance oriented computing would be very helpful.

The paper is organized as follows. Section 2 briefly describes the UML extension mechanism. Section 3 shows how to describe computer architectures and process topologies by using UML. In Section 4, we demonstrate how to model some of the most important sequential, distributed memory and shared memory concepts. The mapping of applications to process topologies is described in Section 5. Section 6 discusses related work. Finally, some concluding remarks are made and future work is outlined in Section 7.

2 Background

The UML defines nine diagram types, which allow to describe different aspects of a system: *use case diagram*, *class diagram*, *object diagram*, *statechart diagram*, *sequence diagram*, *collaboration diagram*, *activity diagram*, *component diagram*, and *deployment diagram*. Each diagram type describes a system or parts of it from a certain point of view. For the purpose of modeling performance-oriented distributed and parallel applications we concentrate on a subset of the UML that consists of class, activity and collaboration diagrams. In this section we present some background information that will be helpful to understand the remainder of this paper. A comprehensive discussion on UML can be found in [11].

Our approach relies on the UML extension mechanisms to customize UML for the domain of performance oriented parallel and distributed computing. The

UML extension mechanisms [7] describe how to customize specific UML model elements and how to extend them with new semantics by using stereotypes, constraints, tag definitions, and tagged values.

Stereotypes are used to define specialized model elements based on a core UML model element. A stereotype refers to a base class in the UML metamodel (see Fig. 1.a), which indicates the element to be stereotyped. A stereotype may introduce additional values, additional constraints, and a new graphical representation. Stereotypes are notated by the stereotype name enclosed in guillemets $<< \ldots >>$ or by a graphic icon. We are employing stereotypes to define modeling elements for constructs such as SEND, RECEIVE, PARALLEL, CRITICAL, etc. UML properties – in form of a list of *tag-value* pairs – are introduced to attach additional information to modeling elements. A tag represents the name of an arbitrary property with a given value and may appear at most once in a property list of any modeling element. It is recommended to define tags within the context of a stereotype. The notation of tags follows a specific syntax: {tag = value}, for instance, {time=10}. A *constraint* allows to linguistically specify new semantics for a model element by using expressions in a designated constraint language. Constraints are specified as a text string enclosed in braces { }, for instance, {WaitUntilCompleted = True}.

The usage of the UML extension mechanisms is illustrated in Fig. 1. Figure 1.a depicts the definition of the modeling element *action+* by stereotyping the base class *ActionState*. An *ActionState* is used to model a step in the execution of an algorithm. The compartment of the stereotype *action+* named *Tags* specifies a list of tag definitions which includes *id*, *type*, and *time*. Tag *id* can be used to uniquely identify the modeling element *action+*; tag *type* specifies the type of *action+*, and tag *time* the time spent to complete *action+*. We are using *action+* (see example in Fig. 1.b) to model various types of single-entry single-exit code regions, whereas tags are employed to describe performance relevant information, such as estimated or measured execution times (cf. Fig. 1.b). The set of the tag definitions is not limited to those shown in Fig. 1.a but can be arbitrarily extended to suffice a modeling objective.

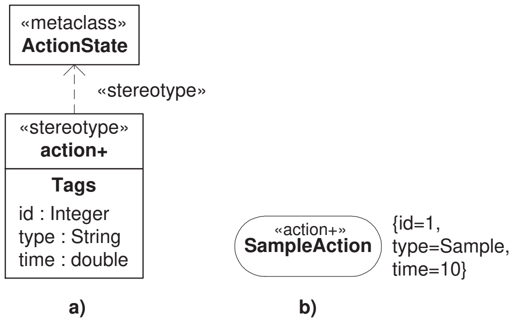


Fig. 1. Definition of stereotype action+

In the remainder of this paper, programming language constructs will be denoted with capital letters (e.g., non-blocking SEND) and UML modeling elements with small letters (e.g., *nbsend*). For the sake of simplicity, in some examples the properties of a modeling element are suppressed.

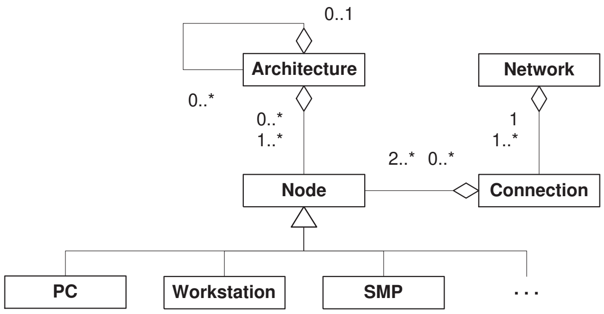


Fig. 2. UML class diagram of distributed and parallel architectures

3 Modeling Computer Architectures and Process Topologies

A distributed/parallel machine can be described as a set of computational nodes, such as single processor nodes (for example, PC or workstation), symmetric multiprocessing systems (SMP - consists of multiple processors that share a common memory), etc. These computational nodes are interconnected by one or several specific networks (see Fig. 2). Distributed and parallel architectures can be represented straightforwardly by a UML deployment diagram (see Section 5). Commonly in parallel processing programs are not directly mapped onto a physical architecture but first onto a virtual architecture (process topology) that can be of arbitrary size. The mapping of virtual to physical architectures is done at runtime which commonly distributes several processes of the virtual architecture onto a single physical processor in order to honor the actual size of a physical architecture. In the following section we outline how to model process topologies with UML collaboration diagrams.

A process topology may be defined as a group of processes that have a predefined regular interconnection topology such as a *farm*, *ring*, *2D mesh* or *tree* which is described in [10]. The description of a process topology assumes a virtual architecture of arbitrary size (number of processing units). The virtual architecture can be mapped on a physical architecture lateron, e.g. for the sake of simulating the performance behavior of a program on a specific target architecture. In the remainder of this paper we will use the term *processing unit* for process or thread when is appropriate.

4 Modeling Distributed and Parallel Applications

In this section we describe our approach of modeling performance-oriented distributed and parallel applications by using activity diagrams. The basic idea is to specify a set of building blocks (see Fig. 3) that represent key concepts of sequential, shared memory, and message passing constructs and thus allow to model basically arbitrary large and complex applications when grouped together. The building blocks have been largely motivated by OpenMP [3] and MPI [12] standards. OpenMP is a specification for a set of compiler directives, library routines, and environment variables that is used to specify shared memory parallelism in Fortran and C/C++ programs. MPI is a library of routines that supports message passing programming in Fortran and C. For each building block we will describe how it can be captured using fragments of activity diagrams and discuss problems or open issues that emerged during our work. Because of limited space we will focus on some of the most interesting building blocks in this paper. For additional building blocks (Fig. 3) the reader may refer to [10, 9].

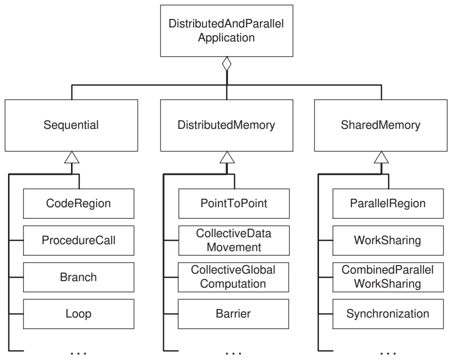
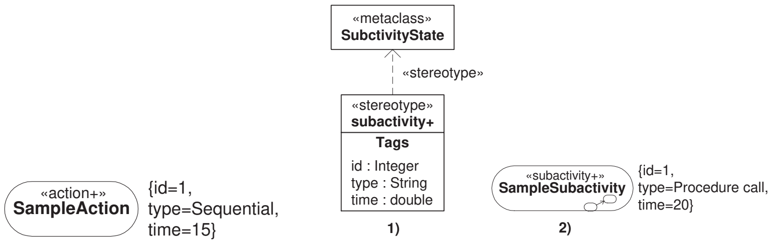


Fig. 3. The overview of concepts

The majority of performance-oriented scientific applications are implemented using procedural languages. In contrast to object oriented languages, procedural programs are control flow oriented. For this reason, we have selected the UML activity diagram in order to model computational, communication, and synchronization operations of an application.

We discovered several problems during our effort to model parallel and distributed applications with activity diagrams. The current UML 1.4 standard describes activity diagrams as a variation of a statechart diagram. Several constructs of activity diagrams lack a precise syntax and semantics. For instance, the well-formedness rules linking forks with joins are not fully defined, nor are the concepts of dynamic invocation and deferred events, among others. For our modeling purpose, we require to associate additional information (e.g. performance and scheduling data) with activities. The UML 1.4 specification [7] recommends that tag definitions should be defined in the context of a stereotype. As a consequence, we introduced stereotypes for many building blocks in order to associate additional information to activities. The usage of stereotypes also alleviated or avoided complex representation of some parallel and distributed program patterns (e.g. BROADCAST).



(a) SEQUENTIAL code region (b) PROCEDURE CALL

Fig. 4. Modeling a SEQUENTIAL code region and a PROCEDURE CALL

4.1 Sequential Concepts

A sequential construct is used to model the unit of work that is executed by a single processing unit (process or thread). In what follows, we describe a few sequential concepts in order to outline our modeling approach.

The stereotypes *action+* and *subactivity+* are used to model various types of single-entry single-exit code regions. The UML modeling elements ActionState (see Fig. 1.a) and SubactivityState (see Fig. 4.b.1) have been stereotyped in order to add properties that are relevant, for performance modeling. Figure 4.a illustrates how to model SEQUENTIAL code regions. The property *type* of the stereotype *action+* indicates that the modeled code region is executed in sequential mode.

In Fig. 4.b.1 we introduce the stereotype *subactivity+* whose purpose in this case is to represent a PROCEDURE CALL. The property *type* of the stereotype *subactivity+* indicates that the modeled code region is a procedure call.

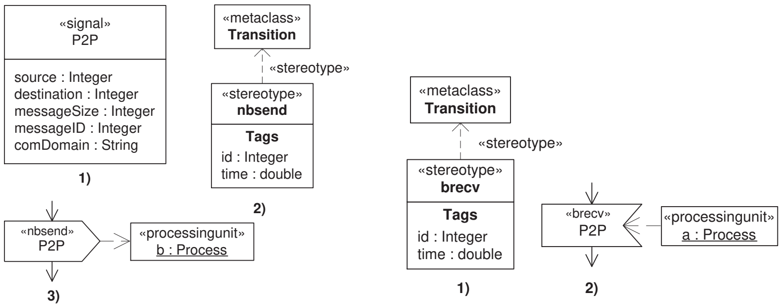
4.2 Distributed Memory Concepts

The most commonly used programming method for distributed memory architectures is dominated by the message passing model. Multiple processing units are created each of which is identified by an unique name. Processing units can employ *point-to-point* communication operations to send messages from a named processing unit to another. A group of processing units can use *collective* communication operations to perform global communication.

In our modeling approach the *communication event* holds the information about the pattern of communication (for example point-to-point), the communication size (number of processes), and the communication domain (process group). A communication event is associated with a transition in an activity diagram. For instance, a communication event RECEIVE triggers a transition into an action state.

Point-to-point Communication (P2P) is commonly implemented by using SEND and RECEIVE operations. P2P operations can be classified into blocking and non-blocking operations. A non-blocking SEND initiates the send operation without being blocked at the sender. As soon as the send of a message is initiated, the sender can proceed with the execution of the program. For the definition of the signal event P2P we use the predefined UML stereotype *signal* (see Fig. 5.a.1) and its parameters (*source*, *destination*, *messageSize*, etc.). A non-blocking SEND is modeled by the stereotype *nbsend*. Figure 5.a.2 illustrates the definition of the stereotype *nbsend* based on the base class *Transition*. An example usage of *nbsend* is presented in Fig. 5.a.3 where a signal *P2P* is sent to the processing unit *b:Process*.

A blocking RECEIVE waits until the receive buffer contains the message received which is modeled as a new stereotype denoted as *brecv*. Figure 5.b.1 presents the definition of the stereotype *brecv* based on the base class *Transition*. Figure 5.b.2 depicts the receiving of a signal *P2P* from the processing unit *a:Process*.



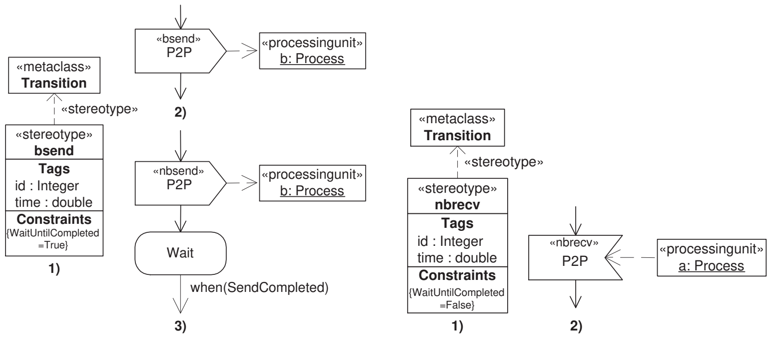
(a) Non-blocking SEND (b) Blocking RECEIVE

Fig. 5. Modeling non-blocking SEND and blocking RECEIVE

For representing non-blocking SEND and blocking RECEIVE operations we have used the adequate UML control icons. Although it should be mentioned, that using these icons is not necessary, because non-blocking SEND and blocking RECEIVE operations can be specified on transitions. However, we prefer to employ these icons because they improve the readability of UML models.

On the other hand, application developers for parallel and distributed systems extensively use blocking SEND and non-blocking RECEIVE operations, for which UML does not provide a specific notation. In order to alleviate this drawback, we propose to use the same SEND and RECEIVE control icons provided by the UML standard but with appropriate stereotypes in order to define their exact meaning. A blocking SEND is not completed until the message has been sent such that the sender can access and overwrite the send buffer without side effects. A blocking SEND is modeled by the stereotype *bsend*. Figure 6.a.1 illustrates the definition of the stereotype *bsend* based on the base class *Transition*. The constraint {WaitUntilCompleted=True} specifies that transition to the next state is deferred until the SEND operation is completed. An example usage of *bsend* is presented in Fig. 6.a.2 where a signal *P2P* is sent to the processing unit *b:Process*. Note that a blocking SEND can be modeled by using the stereotype *nbsend* (see Fig. 6.a.3) and a *Wait* state. After the SEND operation – modeled by the stereotype *nbsend* – is initiated, the transition to the next action state is deferred until the SEND is completed. Waiting for completion of the SEND can be modeled by the *Wait* state and the event *when(SendCompleted)*. However, in order to improve the readability of the resulting UML model, we prefer to represent a blocking send by an icon as depicted in Fig. 6.a.2.

A non-blocking RECEIVE initiates the receive operation without waiting for completion. We introduce the stereotype *nbrecv* (see Fig. 6.b.1) which is defined based on base class *Transition*, to model non-blocking RECEIVE operations. The constraint {WaitUntilCompleted=False} indicates that the transition to the next state is not deferred until the RECEIVE operation is completed. An example usage of *nbrecv* is presented in Fig. 6.b.2 where a signal *P2P* is received from the processing unit *a:Process*.

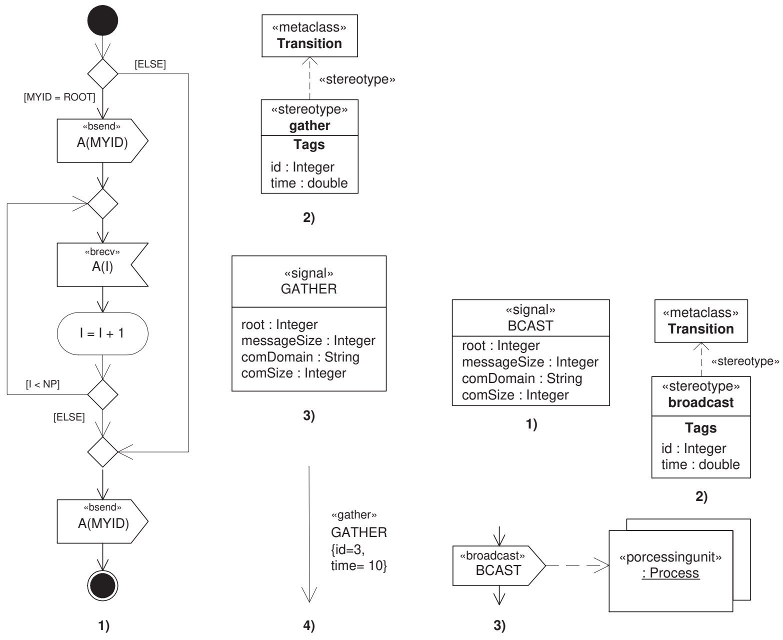


(a) Blocking SEND (b) Non-Blocking RECEIVE

Fig. 6. Modeling blocking SEND and non-blocking RECEIVE

Collective Communication involves communication among a group of processing units to move data or to conduct global computation. In order to represent collective communication operations we encountered difficulties to model and visualize a processing unit that sends/receives several messages to/from a set of processing units. Our goal is to find a notation that improves the understanding of the model and still expresses the concept in a compact form. One possible option to model collective communication could involve a loop that includes multiple send/receive operations. This approach results in implementation dependent and rather complex models (see Fig. 7.a.1). In our opinion a better solution to model collective operations is to use stereotypes that hide the implementation of collective communications. For the sake of illustration, we employ the BROADCAST collective communication construct that sends a message to all processing units of a group. A new stereotype denoted as *broadcast* is introduced to model the BROADCAST collective communication. Figure 7.b.1 shows the definition of the event BCAST which is based on the predefined UML stereotype *signal*. Figure 7.b.2 presents the definition of the stereotype *broad-*

cast based on the base class *Transition*. An example of stereotype *broadcast* is depicted in Fig. 7.b.3 where a signal BCAST is sent to a group of processing units.



(a) Modeling GATHER (b) Modeling BROADCAST

Fig. 7. Modeling collective communication concepts

Commonly message passing paradigms also support a GATHER collective communication construct where every processing unit involved in the GATHER operation (see Fig. 7.a.1), sends a message to a single receiver processing unit (called root). Every processing unit is identified by a unique identification (MYID). The root, for which the condition MYID=ROOT is true, sends the message A(MYID) and receives one message A(I) from every SENDER processing unit. NP is the number of processing units involved. Note that the root is also sending a message to itself.

The UML model of the GATHER collective communication construct can be used to present specific implementations of a GATHER operation as shown in Fig. 7.a.1. But in many cases, it is sufficient to specify that a GATHER operation is conducted without including a detailed implementation model. For this purpose, we introduce a new stereotype named *gather* to model GATHER operation in a more compact form (see Fig. 7.a.2). Figure 7.a.3 shows the definition of the event GATHER. An example of stereotype *gather* is depicted in Fig. 7.a.4 where GATHER denotes an event whose occurrence invokes the transition, and {id=3, time=10} indicate example tag values of the stereotype *gather*.

4.3 Shared Memory Concepts

In the shared memory programming model processing units share a common address space which they read and write asynchronously. Various mechanisms such as *locks* and *semaphores* are used to control access to the shared memory. *Work sharing* constructs are provided to divide the execution of a code region among the processing units. To illustrate our approach, in the sequel we will show how to model some of the most important shared memory constructs by using fragments of activity diagrams.

Parallel Region A PARALLEL region is a code region that is executed by multiple threads in parallel. The code enclosed within the PARALLEL region is duplicated and all threads will execute it.

A new stereotype named *parallelregion* is introduced in order to model PARALLEL regions. Figure 8.a depicts the definition of the stereotype *parallelregion* based on the base class *SubactivityState*. An example usage of this stereotype is illustrated in Fig. 8.b. The '*' symbol in the upper right corner of a state denotes *dynamic concurrency*, which means that the actions of an action state or the activity graph of a subactivity state may be executed more than once concurrently, for instance, by multiple threads.

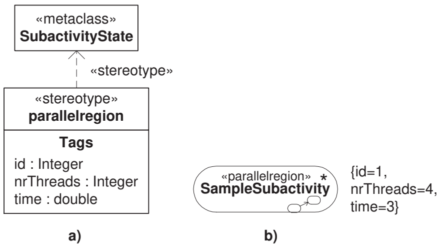
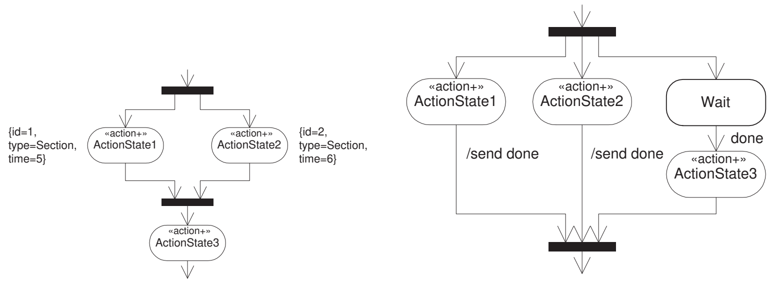


Fig. 8. Modeling the PARALLEL region.

Work-sharing A *work-sharing* construct – must be enclosed within a parallel region – divides the execution of the enclosed code region among the members of the set of threads that encounter it. There is no implied barrier upon entry to a *work-sharing* construct. *Work-sharing* constructs must be encountered by all threads in a set or by none at all. Successive *work-sharing* constructs must be encountered in the same order by all members of a set. Work-sharing constructs do not invoke new threads.

The SECTIONS work-sharing construct specifies that the enclosed sections of code are to be divided among threads in the team. An example with two sections is illustrated in Fig. 9.a. The value of tag *type* specifies that the stereotype *action+* refers to a *section*.

In case that NOWAIT is specified threads that finish early may proceed to instructions following the SECTIONS work-sharing construct without synchronization at the end of the work-sharing construct. The UML model for this case is depicted in Fig. 9.b. As soon as *ActionState1* or *ActionState2* – each of which



(a) SECTIONS work-sharing (b) In case that NOWAIT is specified

Fig. 9. Modeling SECTIONS work-sharing

represent a section – is completed, a signal is produced which causes the *Wait* state to be exited and a transition to *ActionState3* to be triggered. Note that *ActionState3* represents a set of instructions following the work-sharing construct SECTIONS. However, we consider this solution to be too complex. Moreover, it is not general, because it cannot be applied when a work-sharing construct SECTIONS is placed in a loop. It is unclear what UML mechanism could be used to tackle this problem. One solution would be to introduce a new modeling element in UML with appropriate syntax and semantics by using the so-called 'heavyweight' extensibility mechanism as defined by the MOF specification [6].

Synchronization The shared memory synchronization construct controls how the execution of each thread proceeds relative to other threads in a team of threads. The CRITICAL construct restricts access to the enclosed code to only one thread at any given time. A thread waits at the beginning of a critical section until no other thread is executing a critical section with the same *name*.

Figure 10.a depicts the definition of the stereotype *critical* based on the base class *SubactivityState* which is used to model the CRITICAL construct. The tag *name* specifies the name of the critical region. An example usage of stereotype *critical* is given in Fig. 10.b.

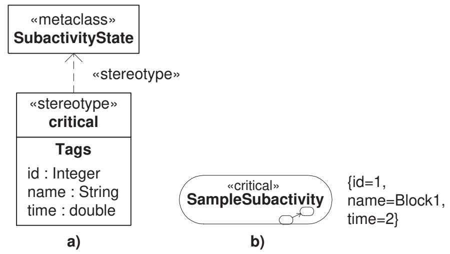


Fig. 10. Modeling the CRITICAL construct

The ATOMIC construct ensures that a specific memory location is updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads. Figure 11 shows an example of modeling the *atomic* construct. The value of the tag *type* specifies that the stereotype *action+* represents an *atomic ActionState*.

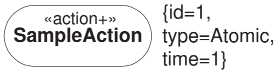


Fig. 11. Modeling the ATOMIC construct.

5 Mapping

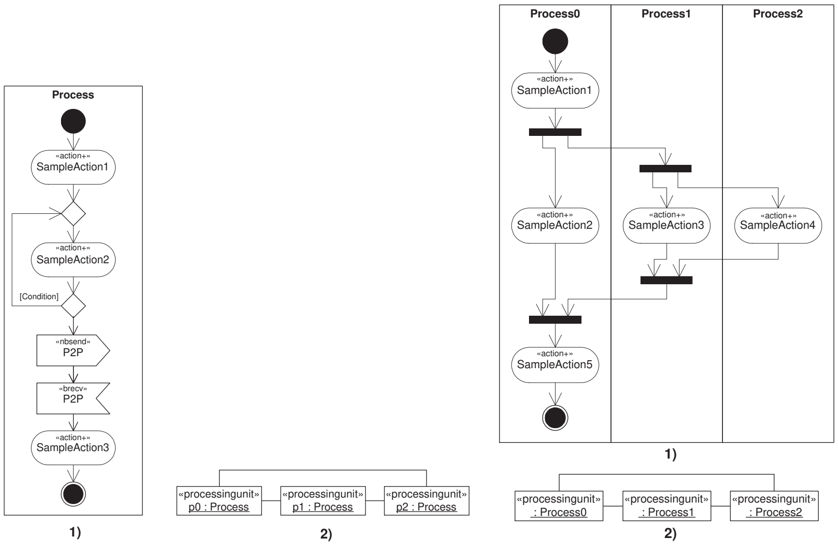
In this section we elaborate how to map an activity diagram to a collaboration diagram in order to reflect the mapping of programs to process topologies. For this purpose we use UML *swimlanes* to define responsibilities of actions and subactivities. An activity diagram may be divided visually into *swimlanes*, each separated from neighboring swimlanes by vertical solid lines on both sides. The relative ordering of the swimlanes has no semantic significance. Each action is assigned to one *swimlane*.

Figure 12 illustrates how a single processor multiple data (SPMD) application is mapped onto a process topology. The activity diagram that represents the program is located within the *swimlane* named *Process* (see Fig. 12.a.1) which is responsible for all activities of the program. The application associated with the *swimlane Process* is mapped to each processing unit of the ring process topology (see Fig. 12.a.2).

Figure 12.b illustrates the mapping of a multiple instruction multiple data (MIMD) application. The *swimlane Process0* processes *SampleAction1*, *SampleAction2*, and *SampleAction5*. *Swimlanes Process1* and *Process2* are, respectively, responsible for *SampleAction3* and *SampleAction4* (see Fig. 12.b.1). The mapping of swimlanes to a ring process topology is depicted in Fig. 12.b.2.

The mapping of process topologies to the physical topology of a parallel and distributed computer architecture is straightforward. It can be accomplished by using the UML deployment diagram to represent the physical topology of the computer architecture. Deployment diagrams visualize the configuration of physical processing elements and the software components that execute on the processing elements [7]. The nodes in a deployment diagram represent processing resources and are are visualized as cubes. Connections between the nodes are shown as arcs joining the nodes. A unit of a software in source, binary, or executable form is represented by a component. A component is denoted as a rectangle with two small rectangles protruding from its side.

Figure 13 shows three nodes interconnected by a network. The component that resides at a node is displayed within a node. The component is used to represent the process and its associated form (source, binary, or executable)



(a) SPMD based mapping (b) MIMD based mapping

Fig. 12. Mapping for SPMD and MIMD applications

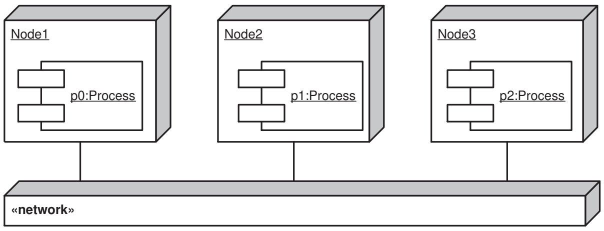


Fig. 13. Mapping of a process topology to a physical topology by the UML deployment diagram

mapped to the node. In this example the network is represented by a node. This kind of notation is used when more than two computer nodes are connected by a network.

6 Related Work

In this section we provide an overview of related work, focusing on application representation issues for performance-oriented parallel and distributed applications.

The POEMS [1] project introduced a graphical representation which captures the parallel structure, communication, synchronization, and sequential behavior of the application. No standards are used to model parallel applications and we have not found any document that describes a graphical representation of process topologies or computer architectures.

The PlusPyr project [2] introduced a parameterized task graph which is a representation of some commonly used directed acyclic graphs. This representation is automatically generated based on an existing annotated sequential program to support program parallelization for the message passing paradigm. In contrast, our representation does not assume the existence of a code and goes beyond message passing applications. Again, there is no support for a graphical representation of process topologies.

In [8] is presented an automatic transformation of UML models annotated with performance information into layered queuing network (LQN) performance descriptions. Their method has been validated for client server applications, whereas our research targets mostly performance-oriented scientific applications that exploit message passing, shared memory, data parallelism, and mixed forms of parallelism.

7 Conclusion

In this paper we have used the UML extension mechanisms to customize UML for the domain of performance oriented computing. We have presented a set of UML building blocks that model some of the most important concepts of message passing and shared memory parallel paradigms. A flexible mechanism has been described for mapping application models onto process topologies. For space reasons we have not presented all building blocks. Additional building blocks are presented in [10, 9].

We conclude that UML provides a rich set of modeling concepts, notations, and mechanisms to substantially alleviate the understanding, documentation, and visualization of the structure and dynamic behavior of distributed and parallel applications including the mapping of applications onto multiprocessor architectures. UML is a widely used standard and, therefore, offers a substantial advantage compared to approaches that use ad-hoc or self-defined graphical representations. Application developers that rely on procedural languages are inherently focused on the control flow of a program which is very well supported by UML activity diagrams. Moreover, mapping of applications onto process topologies can be easily expressed through UML collaboration diagrams.

We determined that by using the core UML, a substantial number of important shared memory and message passing concepts can not be modeled at all or results in complex diagrams. Combining several UML modeling elements with a complex control flow can easily disconcert and conceal the essence of a higher level concept to be represented. For this reason we use a rather large number of stereotypes to describe individual but important concepts of parallel and distributed programming paradigms. Stereotypes enable the definition of higher level concepts (e.g. parallel or critical code regions) with simple notations. Tags can be used to associate arbitrary information. However, it remains an open issue how to describe the parallel SECTION work sharing construct under UML which is a key concept in OpenMP (industry standard for shared memory programming). For this specific concept we tend to propose a new modeling element by incorporating the so-called 'heavyweight' UML extensibility mechanism as defined by the MOF specification.

In summary, we observe that the core UML does not sufficiently support modeling of performance-oriented parallel and distributed applications. Therefore, we believe that the definition of a UML profile for the domain of performance oriented computing is very much desirable and can avoid development of various UML extensions for this domain used by different groups. Currently, we are in the progress to develop a performance estimation tool that uses UML to model parallel and distributed applications [10]. We are building a simulator that determines the performance behavior for UML models enriched with performance information for a range of cluster architectures [4]. The objective is to provide the user with performance information at an early development stage of an application.

References

1. V. Adve, R. Bagrodia, J. Browne, E. Deelman, A. Dube, E. Houstis, J. Rice, R. Sakellariou, D. Sundaram-Stukel, P. Teller, and M. Vernon. POEMS: End-to-End Performance Design of Large Parallel Adaptive Computational Systems. *IEEE Transactions on Software Engineering*, 26:1027–1048, November 2000.
2. M. Cosnard and M. Loi. Automatic task graph generation techniques. In Hesham El-Rewini and Bruce D. Shriver, editors, *Proceedings of the 28th Annual Hawaii International Conference on System Sciences. Volume 2: Software Technology*, pages 113–122, Los Alamitos, CA, USA, January 1995. IEEE Computer Society Press.
3. L Dagum and R Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, January/March 1998.
4. T. Fahringer and S. Pllana. Performance Prophet, Institute for Software Science, University of Vienna, 2002. http://www.par.univie.ac.at/project/prophet
5. P. Kacsuk, G. D'ozsa, and T. Fadgyas. Designing Parallel Programs by the Graphical Language GRAPNEL. *Microprocessing and Microprogramming*, 41:625–643, 1996.
6. OMG. Meta Object Facility (MOF) Specification. http://www.omg.org, November 2001.
7. OMG. Unified Modeling Language Specification. http://www.omg.org, September 2001.
8. D. Petriu and H. Shen. Applying the UML Performance Profile: Graph Grammar-based Derivation of LQN Models from UML Specifications. In *Performance TOOLS 2002, LNCS 2324, Springer-Verlag*, London, UK, April 2002.
9. S. Pllana and T. Fahringer. Modeling Parallel Applications with UML. In *15th International Conference on Parallel and Distributed Computing Systems (PDCS 2002)*, The Galt House, Louisville, KY USA, September 19 - 21 2002.
10. S. Pllana and T. Fahringer. UML Based Modeling of Performance Oriented Parallel and Distributed Applications. In *Proceedings of the 2002 Winter Simulation Conference, E. Ycesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes, eds.*, San Diego, California, USA, December 8-11 2002.
11. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.
12. M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: the complete reference*. MIT Press, Cambridge, MA, USA, 1996.