Modeling Parallel Applications with UML

S. Pllana
Institute for Software Science
University of Vienna
Vienna, 1090, AUSTRIA
pllana@par.univie.ac.at

T. Fahringer
Institute for Software Science
University of Vienna
Vienna, 1090, AUSTRIA
tf@par.univie.ac.at

Abstract

The Unified Modeling Language (UML) offers an extensive set of diagrams for modeling. However, the semantics of specific diagrams aren't always clear so as to decide how to model specific aspects of parallel applications. In this paper we introduce an approach for modeling parallel applications with UML by utilizing the UML extension mechanisms. We will demonstrate that these extension mechanisms represent a powerful means to model a wide range of structural and behavioral patterns of the parallel programming paradigm at various levels of details by modeling some of the most important parallel programming concepts.

1 INTRODUCTION

Usually the cost of the development of parallel applications is very high. Without proper design a mismatch might occur between the proposed system and the actual system developed. For many years, researchers have tried to use visual modeling in order to improve the understanding of complex software systems. However, previous work has been only moderately successful because it invented new modeling languages that were not based on a standard.

With the appearance of the Unified Modeling Language (UML), 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. UML is widely used for domains such as telecommunications [7], aviation industry [12], transportation industry [15], business software systems [6], real time systems [14], and distributed web applications [2]. Furthermore, XML Metadata Interchange (XMI) [10] is a standard storage representation for UML diagrams approved by Object Management Group (OMG).

The benefits of using UML in the domain of parallel computing can be manifold: (i) documentation and visualization of existing applications (ii) specification, visualization, construction, and documentation of new applications, (iii) performance modeling.

Unfortunately so far there was very little work which addressed the issue of how to use UML for modeling parallel applications. UML offers an extensive set of diagrams for modeling and even the semantics of specific diagrams aren't always clear so as to decide how to model specific aspects of parallel programs. In order to overcome these problems, in this paper we introduce basically the first approach for modeling parallel applications based on the UML.

We utilize the UML extension mechanisms to customize UML for the domain of parallel computing. We are using only a small subset of UML, namely class, activity, and collaboration diagrams. Class diagrams are used to build the structural model of 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 are describing a set of UML building blocks that model some the most important constructs of shared memory paradigm. The building blocks have been largely motivated by Open Multi Processing (OpenMP) [4] standard. Moreover, these building blocks can be annotated with arbitrary information, such as performance data, which is very useful to meet individual modeling needs. Based on the UML building blocks we can develop models for large and complex applications.

A critical look at the utility of the UML to model performance oriented shared memory and message passing applications is provided in [11]

The paper is organized as follows. Section 2 describes the subset of UML that is relevant for modeling of parallel applications. In the next section, we show how to model the most important sequential and shared memory concepts. Section 4 discusses related work. Finally, some concluding remarks are made and future work is outlined in Section 5.

2 BACKGROUND

The UML defines nine diagram types, which allow different aspects of a system to be expressed: *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 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 [13]. In what follows we briefly describe the UML activity diagram, and UML extension mechanisms.

An *activity diagram* is a variation of a state machine in which the states represent the execution of *actions* or *subactivities* and the transitions are triggered by the completion of the actions or subactivities. We use activity diagrams to model computational, communication, and synchronization operations of parallel applications. An *action* state is used to model a step in the execution of an algorithm, or a workflow process. A *subactivity* state invokes an activity diagram. When a subactivity state is entered, the activity diagram nested in it is executed. A subactivity state is shown in the same way as an action state with the addition of an icon in the lower right corner depicting a nested activity diagram (see Fig. 1.b). *Transitions* are used to specify that the flow of control pass from one action to the next action state. A *wait* state performs no activity, but waits for an event occurrence. An *event* is an occurrence that may trigger a state transition. An activity diagram expresses a *decision* when guard *conditions* are used to indicate different possible transitions (see Fig. 1.a). A guard *condition* specifies a condition that must be satisfied in order to enable the firing of an associated transition. A *merge* has two or more incoming transitions and one outgoing transition. It can be used to merge decision branches back together. *Fork* and *join* are used to model parallel flows of control (see Fig. 1.b). The *initial* and *final* state are, respectively, visualized as a solid ball and a solid ball inside a circle.

Figure 1.a illustrates how to model a loop by employing an activity diagram, whereas Fig. 1.b shows one option for modeling the parallel execution of two code regions.

Our approach relies on the UML extension mechanisms to customize UML for the domain parallel computing. The UML extension mechanisms [9] describe

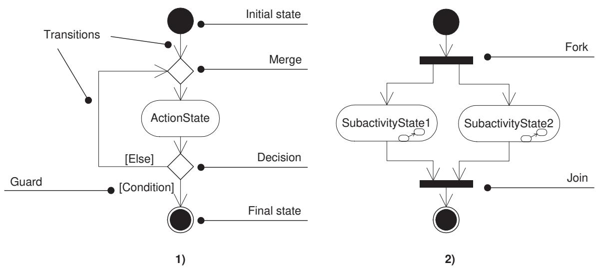

Figure 1: UML activity diagram notation.

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. 2.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 << ... >> or by a graphic icon. We are employing stereotypes to define modeling elements for constructs such as CRITICAL, ATOMIC 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 UML extension mechanisms is illustrated in Fig. 2. Figure 2.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. 2.b) to model various types of single-entry single-exit code

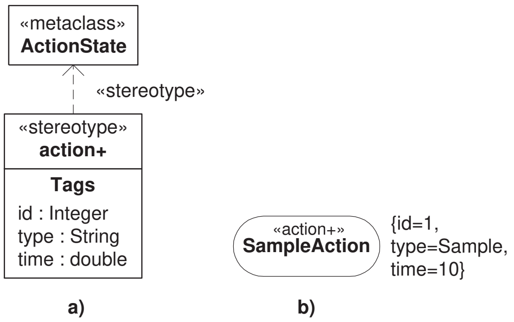

Figure 2: Definition of stereotype action+

regions, whereas tags are employed to describe performance relevant information, such as estimated or measured execution times (cf. Fig. 2.b). The set of the tag definitions is not limited to those shown in Fig. 2.a but can be arbitrarily extended to suffice a modeling objective.

In the remainder of this paper, programming language constructs will be denoted with capital letters, for instance CRITICAL, and UML modeling elements with small letters, for instance *critical*. For the sake of simplicity in some examples the properties of a modeling element are suppressed.

3 MODELING PARALLEL APPLICATIONS

In this section we describe how to use UML for modeling parallel applications. The basic idea is to specify a set of building blocks (see Fig. 3) that represent key concepts of sequential and shared memory constructs and thus allow to model basically arbitrary large and complex applications when grouped together. The building blocks have been largely motivated by Open Multi Processing (OpenMP) [4] standard. 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.

3.1 Sequential Concepts

A sequential construct is used to model an unit of work that is executed by a single processing unit (process or thread). In what follows, we described 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. 2.a) and SubactivityState (see Fig. 5.a) have been stereotyped in order to add properties that are

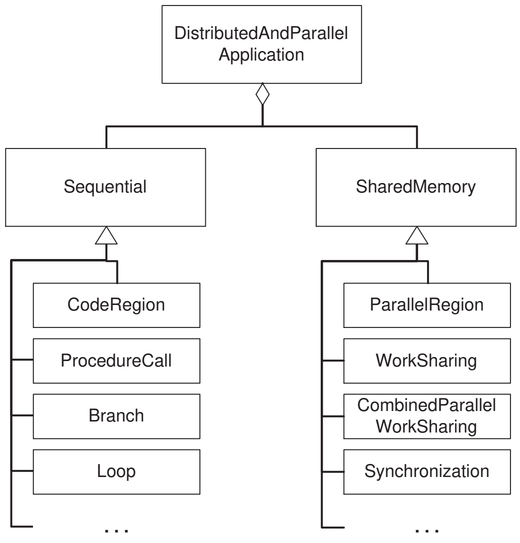

Figure 3: The overview of concepts.

relevant, for instance, for performance modeling. Figure 4 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.

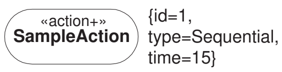

Figure 4: Modeling a SEQUENTIAL code region.

In Fig. 5.a 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.

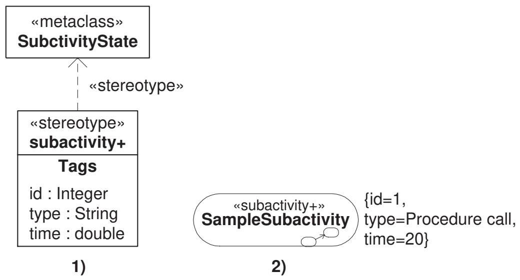

Figure 5: Modeling a PROCEDURE CALL.

3.2 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* may be used to control access to the shared memory. *Work sharing* constructs may be used 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.

3.2.1 Parallel Region

A PARALLEL region is a code region that is to be 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 6.a depicts the definition of the stereotype *parallelregion* based on the base class *SubactivityState*. Figure 6.b illustrates an example usage of the stereotype *parallelregion*. 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.

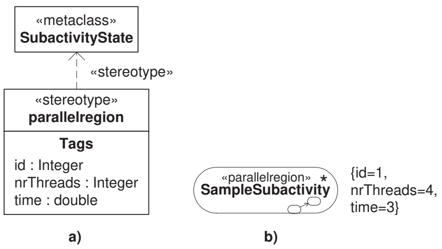

Figure 6: Modeling the PARALLEL region.

A *nested parallel region* is a PARALLEL region nested within another PARALLEL region. When a thread in a team of threads executing a PARALLEL region encounters another parallel region, it creates a new team, and it becomes the master of that new team. Figure 7.a shows an OpenMP code of a nested parallel region example. The PARALLEL region R2 is nested within the PARALLEL region R1. The corresponding UML model is shown in Fig. 7.b.

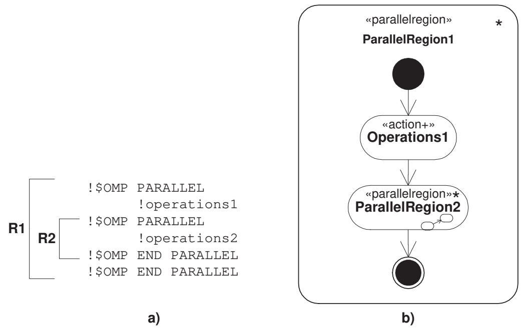

Figure 7: Modeling the nested parallel region.

3.2.2 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 launch new threads.

The DO work-sharing construct specifies that the iterations of the immediately following *do loop* must be executed in parallel. The iterations of the *do loop* are distributed across threads that already exists in a parallel region. We define a new stereotype *doworksharing* in order to specify DO work-sharing constructs (see Fig. 8.a). The tag *scheduleType* indicates how the loop iterations are distributed onto a set of threads. An example is given in Fig. 8.b. Figure 8.c shows the content of subactivity *SampleSubactivity* which represents a loop with an sample action in its body.

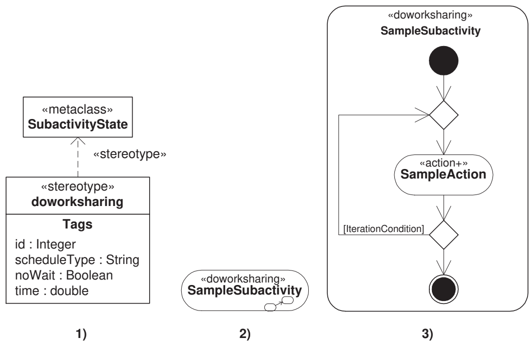

Figure 8: Modeling the DO work-sharing construct.

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. The value of tag *type* specifies that the stereotype *action+* refers to a *section*.

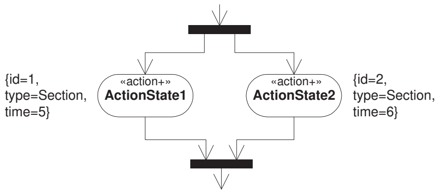

Figure 9: Modeling the SECTIONS work-sharing construct.

3.2.3 Combined Parallel Work-sharing

The combined parallel work-sharing constructs are shortcuts for specifying a parallel region that contains only one work-sharing construct.

The PARALLEL DO construct provides a shortcut form for specifying a PARALLEL region that contains a single DO construct.

The new stereotype named *paralleldo* is defined in order to model the PARALLEL DO construct. Figure 10.a depicts the definition of the stereotype *paralleldo* by using the base class *SubactivityState*. An example of the *paralleldo* stereotype is outlined in Fig. 10.b.

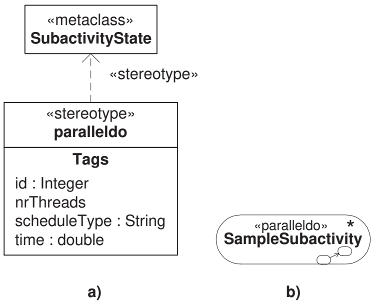

Figure 10: Modeling the PARALLEL DO construct.

3.2.4 Synchronization

The shared memory synchronization construct controls how the execution of each thread proceeds relative to other threads of the team. 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 11.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. 11.b.

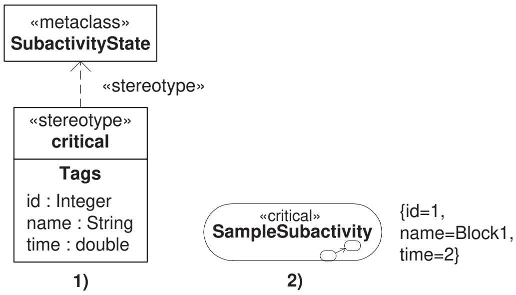

Figure 11: Modeling the CRITICAL construct.

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 12 shows an example of modeling the *atomic* construct. The value of the tag *type* specifies that the stereotype *action+* represents an *atomic ActionState*.

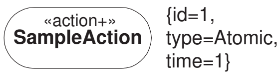

Figure 12: Modeling the ATOMIC construct.

4 RELATED WORK

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

GRAPNEL [8] is a hybrid programming language that uses both graphical and textual representations to describe message passing programs and a graphical representation for process topologies. GRAPNEL introduced a self-defined graphical representation without using any standard such as UML.

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 [3] introduced a parameterized task graph which is a representation of some commonly used directed acyclic graphs. Again, there is no support for a graphical representation of process topologies.

5 CONCLUSION

The emerging of the Unified Modeling Language (UML) as the de facto standard visual modeling language, has unclosed new opportunities for modeling of parallel systems.

In this paper, we have described a set of UML building blocks that model some of the most important constructs of shared memory parallel paradigms which can be used to develop models for large and complex parallel applications.

Currently we are in the progress to develop a performance estimation tool that uses UML to model parallel and distributed applications [5]. We are building a simulator that determines the performance behavior for UML models enriched with performance information for a range of cluster architectures. The objective is to provide the user with performance information at an early development stage of an application.

Acknowledgements

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

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] J. Conallen. Modeling Web Application Architectures with UML. *Communications of the ACM*, 42(10):63–70, October 1999.

[3] 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.

[4] Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, January/March 1998.

[5] T. Fahringer and S. Pllana. Performance Prophet, Institute for Software Science, University of Vienna, 2002. http://www.par.univie.ac.at/project/prophet

[6] L. Hayashi and J. Hatton. Combining UML, XML and Relational Database Technologies. The Best of All Worlds For Robust Linguistic Databases. In *IRCS Workshop on Linguistic Databases*, pages 115–124, University of Pennsylvania, Philadelphia, USA, December 2001.

[7] E. Holz. Application of UML within the Scope of new Telecommunication Architectures. In *GROOM Workshop on UML*, Mannheim, 1997. PhysicaVerlag.

[8] P. Kacsuk, G. D'ozsa, and T. Fadgyas. Designing Parallel Programs by the Graphical Language GRAPNEL. *Microprocessing and Microprogramming*, 41:625–643, 1996.

[9] OMG. Unified Modeling Language Specification. http://www.omg.org, September 2001.

[10] OMG. OMG XML Metadata Interchange (XMI) Specification. http://www.omg.org, January 2002.

[11] S. Pllana and T. Fahringer. On Customizing the UML for Modeling Performance-Oriented Applications. In *<<UML>> 2002, "Model Engineering, Concepts and Tools", Springer-Verlag*, Dresden, Germany, September 30 - October 4 2002.

[12] E. Roberts, J. Villani, M. Osman, D. Godso, B. King, and M. Ricciardi. Aviation System Analysis Capability Executive Assistant Design. Technical Report NASA/CR-1998-207679, Logistics Management Institute, McLean, Virginia, May 1998.

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

[14] B. Selic and J. Rumbaugh. Using UML for Modeling Complex Real-Time Systems. http://www.rational.com/products/whitepapers/100230.jsp, March 1998.

[15] M. Walther, J. Schirmer, P. Torre Flores, A. Lapp, T. Bertram, and J. Petersen. Integration of the ordering concept for vehicle control systems CARTRONIC into the software development process using UML modeling methods. In *SAE 2001 World Congress*, Detroit, Michigan, USA, 2001.