-

UML BASED MODELING OF PERFORMANCE ORIENTED PARALLEL AND DISTRIBUTED APPLICATIONS

Sabri Pllana
Thomas Fahringer

Institute for Software Science
University of Vienna
Vienna, A-1090, AUSTRIA

ABSTRACT

In this paper we introduce a novel approach for modeling performance oriented distributed and parallel applications based on the Unified Modeling Language (UML). We utilize the UML extension mechanisms to customize UML for the domain of performance oriented distributed and parallel computing. A set of UML building blocks is described that model some of the most important constructs of message passing and shared memory parallel paradigms which can be used to develop models for large and complex parallel and distributed applications. We illustrate our approach by modeling a parallel many-body physics application that combines message passing and shared memory parallelism.

1 INTRODUCTION

Many scientific and engineering problems require high performance computation. The development of performance oriented distributed and parallel applications is a time-consuming, error-prone, and tedious process that involves many cycles of code editing, compiling, executing, and performance analysis. We believe that a visual modeling language can substantially alleviate the development of distributed/parallel applications and improve the understanding of the resulting performance behavior on a given target machine.

UML is emerging as the de facto standard visual modeling language which is general purpose, broadly-applicable, tool-supported, and industry-standardized. UML is widely used for domains such as telecommunications (Holz 1997), transportation industry (Walther et al. 2001), bussines software systems (Hayashi and Hatton 2001), real time systems (Selic and Rumbaugh 1998), and distributed web applications (Conallen 1999). In addition, in (Rossetti et al. 2000) the UML is used for the documentation of a software design framework for object-oriented simulation. Furthermore, XML Metadata Interchange (XMI) (OMG 2002) is a standard storage representation for UML diagrams approved by the Object Management Group (OMG).

The benefits of using UML in the domain of performance-oriented distributed and parallel computing are manifold.

- Documentation and visualization of existing applications: UML is extensively used in many information technology areas to document and to visualize source codes which alleviates the maintenance and further development of existing applications. The old saying "*a picture is worth a thousand words*" is also valid in the domain of parallel and distributed computing.
- Specification, visualization, construction, and documentation of new applications: Software engineering technology hasn't been widely used in the domain of parallel and distributed computing. A graphical representation of an application can be very useful to communicate ideas and describe the design of a complex application at a high level.
- Performance modeling: Based on UML, the application developer can build a performance model of the application at an early development stage. The performance can be predicted and design decisions can still be influenced without time-consuming modifications of large portions of an already implemented application.
- Simulation: Based on a UML model of an application and a simulator for a target architecture, one can predict the execution behavior of the application model. An UML-based performance prediction system is depicted in Figure 1. The user develops the model for an application by composing existing templates (building blocks). The application model is enriched with cost functions. Thereafter, the annotated model is transformed to an intermediate form (e.g. XMI) based on which the simulator examines the behavior for this application on a given target machine model that is selected by the user.

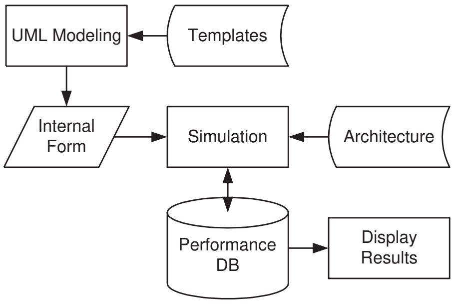


Figure 1: Performance Prediction System

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 and distributed programs. In order to overcome this deficiency, in this paper we describe an approach that utilizes the UML extension mechanisms to customize UML for the domain of performance oriented distributed and parallel computing. We employ only a small subset of UML, namely class, activity, and collaboration diagrams. Class diagrams are used 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 are describing 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 Open Multi Processing (OpenMP) (Dagum and Menon 1998) and the Message Passing Interface (MPI) (Snir et al. 1996) 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. The mixed mode application development may involve other programming languages such as High Performance Fortran (HPF) and POSIX threads, but we focus on OpenMP and MPI because of their portability, prevalence, and the fact that they represent industry standards for shared and distributed memory systems respectively. Note that the proposed UML building blocks represent generic concepts whereas OpenMP and MPI are specific implementations. Moreover, these building blocks can be annotated with arbitrary information, such as performance data. Based on the UML building blocks we can develop models for large and complex applications.

We illustrate our approach by modeling a mixed parallelism many-body physics application which comprises both message passing and shared memory parallelism.

The paper is organized as follows. The next section describes the subset of UML which is used for modeling of distributed and parallel applications. Section 3 depicts how to model process topologies. The following section describes how to model some of the most important sequential, distributed memory and shared memory concepts. A case study is discussed in Section 5. Finally, some concluding remarks are made and future work is outlined in Section 6.

2 BACKGROUND

The UML defines nine diagram types, which allow different aspects of a system to be expressed. 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 (Rumbaugh et al. 1999).

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 (OMG 2001) describes 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 Figure 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 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}.

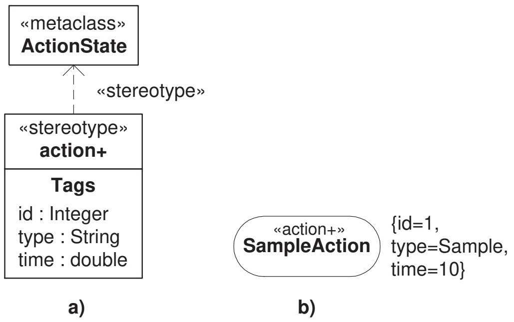


Figure 2: Definition of Stereotype action+

The usage of UML extension mechanisms is illustrated in Figure 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 Figure 2.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. Figure 2.b). The set of the tag definitions is not limited to those shown in Figure 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 non-blocking SEND, and UML modeling elements with small letters, for instance *nbsend*. For the sake of simplicity, in some examples the properties of a modeling element are suppressed.

3 MODELING PROCESS TOPOLOGIES

Distributed and parallel architectures can be represented straightforwardly by a UML deployment diagram which has been described in (Pllana and Fahringer 2002). Commonly, parallel 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 between virtual and physical architecture is done immediately before the parallel program is executed on the target architecture. Commonly, a mapping strategy distributes several processes of the virtual architecture onto a single physical processor in order to honor the actual size of a physical architecture. In this section we outline how to model process topologies with UML collaboration diagrams.

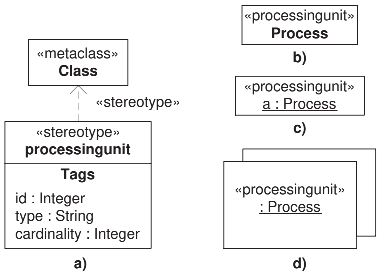


Figure 3: Modeling Processing Units

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*. The description of a process topology is machine-independent and depends only on the application. In order to describe process topologies we introduce the stereotype *processingunit* which is used to model processes or threads. In the remainder of this paper we will use the term *processing unit* for process or thread when appropriate. Figure 3.a illustrates the definition of the stereotype *processingunit* based on the base class *Class* whereby the *cardinality* tag specifies the number of elements in a set of instances. Figure 3.b shows the definition of class *Process* based on the stereotype *processingunit*, which is used to model a process. An instance of class *Process* is represented by the object *a:Process* (see Figure 3.c). The set of instances of the class *Process* is modeled by the multiobject *:Process* (see Figure 3.d). Figure 4 shows several process topology types which are exemplified by *CollaborationInstanceSets*. It comprise a *farm* which is a group of processing units without interconnections (see Figure 4.a), a *ring* which consists of processing units arranged in an interconnected linear array (see Figure 4.b), a 2D mesh that arranges processing units in a two dimensional grid (see Figure 4.c), and a binary tree structure topology (see Figure 4.d).

An alternative representation of a process topology which is well-suited for large number of processing units is depicted in Figure 5.a which defines a stereotype *2d-mesh* by employing the base class *Collaboration*. Figure 5.b illustrates a 2D mesh topology with 3 rows and 3 columns. The mapping of applications to process topologies is described in (Pllana and Fahringer 2002).

4 MODELING APPLICATIONS

In this section we demonstrate how to use UML for modeling of some of the most important sequential, shared memory, and message passing concepts. The basic idea is to specify a set of building blocks (see Figure 6) that represent key concepts of sequential, shared

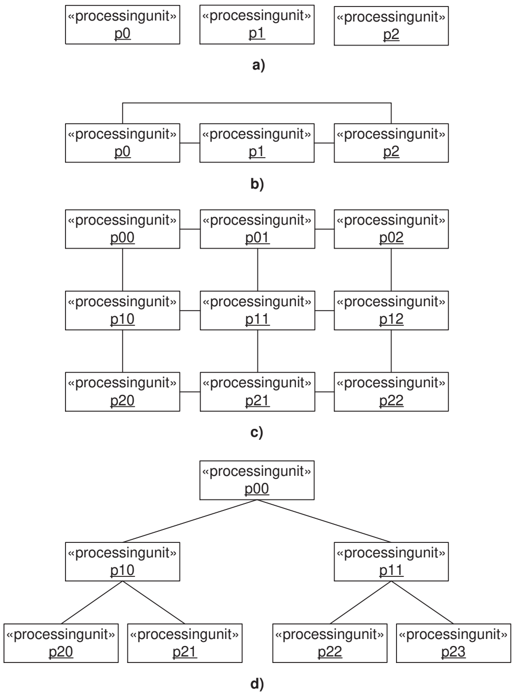


Figure 4: Examples of Process Topologies

memory, and message passing constructs and thus allow to model basically arbitrary large and complex applications when grouped together.

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 Figure 2.a) and SubactivityState (see Figure 9.a) have

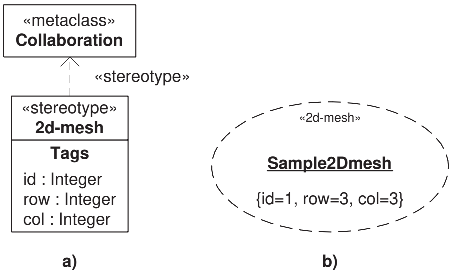


Figure 5: 2D-Mesh (3x3) Process Topology Represented by the 2d-mesh Stereotype

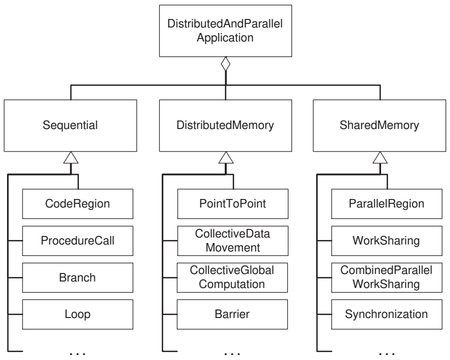


Figure 6: The Overview of Concepts

been stereotyped in order to add properties that are relevant, for instance, for performance modeling. Figure 7 illustrates the modeling of SEQUENTIAL code regions. The property *type* of the stereotype *action+* indicates that the modeled code region is executed in sequential mode.

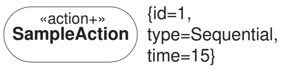


Figure 7: Modeling a SEQUENTIAL Code Region

Figure 8 demonstrates how to model a BRANCH in control flow. Based on the value of a boolean expression *condition*, one of the two alternative control flow paths is selected.

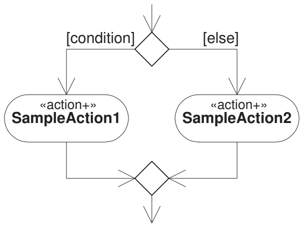


Figure 8: Modeling a BRANCH Control Flow

In Figure 9.a we introduce the stereotype *subactivity+* in order to represent a PROCEDURE CALL. The property *type* of the stereotype *subactivity+* indicates that the modeled code region is a procedure call.

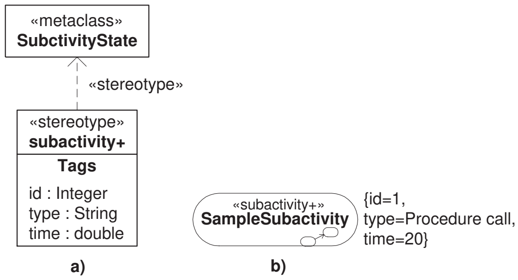


Figure 9: Modeling a PROCEDURE CALL

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 with a unique identifier are created. Interaction among processing units is implied by sending and receiving messages to and from named processing units. Processing units can employ *point-to-point* communication operations to send messages from one named processing unit to another. A group of processing units can use *collective* communication operations to perform global operations among processing units.

In our 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 invokes a transition into an action state.

4.2.1 Point-to-point Communication (P2P)

Exchanging messages via SEND and RECEIVE is the basic mechanism for point-to-point communication which 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 proceeds with the execution of the program. The signal event P2P is modeled by a UML stereotype *signal* (see Figure 10.a) and its attributes: *source*, *destination*, *messageSize*, etc. Figure 10.b illustrates the definition of the stereotype *nbsend* based on the base class *Transition*. An example usage of *nbsend* is presented in Figure 10.c where a signal *P2P* is sent to the processing unit *b:Process*.

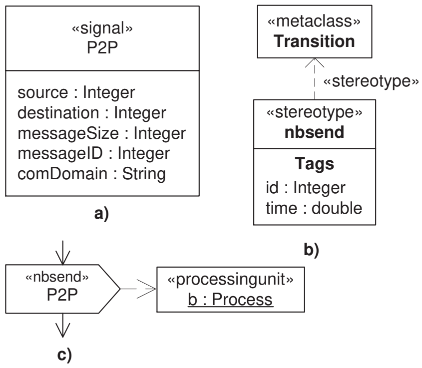


Figure 10: Modeling Nonblocking SEND

A blocking RECEIVE waits until the receive buffer contains the message received which is modeled as a new stereotype named *brecv*. Figure 11.b shows the definition of the stereotype *brecv* based on the base class *Transition*. Figure 11.a. depicts the receiving of a signal *P2P* from the processing unit *a:Process*.

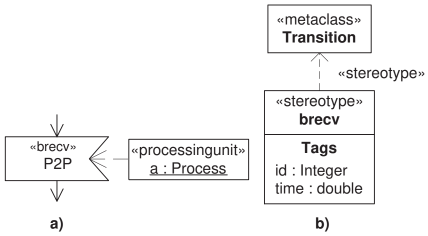


Figure 11: Modeling Blocking RECEIVE

Other point-to-point communication primitives such as blocking SEND and non-blocking RECEIVE operations are modeled analogously by defining adequate stereotypes. Communication that involves a group of processing units is commonly known as collective communication which can be modeled by a set of stereotypes (Pllana and Fahringer 2002).

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* 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 concepts by using fragments of activity diagrams.

4.3.1 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 12.a depicts the definition of the stereotype *parallelregion* based on the base class *SubactivityState*. Figure 12.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.

4.3.2 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

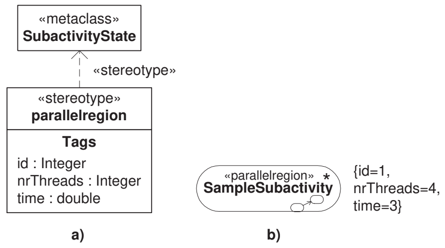


Figure 12: Modeling the PARALLEL Region

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 Figure 13.a). The tag *scheduleType* indicates how the loop iterations are distributed onto a set of threads. An example is given in Figure 13.b. Figure 13.c shows the content of subactivity *SampleSubactivity* which represents a loop with a sample action in its body.

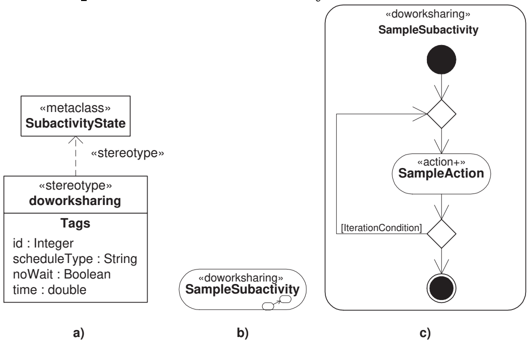


Figure 13: Modeling the DO Work-Sharing Construct

4.3.3 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. A new stereotype named *paralleldo* is defined in order to model the PARALLEL DO construct. Figure 14.a depicts the definition of the stereotype *paralleldo* by using the base class *SubactivityState*. An example of the *paralleldo* stereotype is outlined in Figure 14.b.

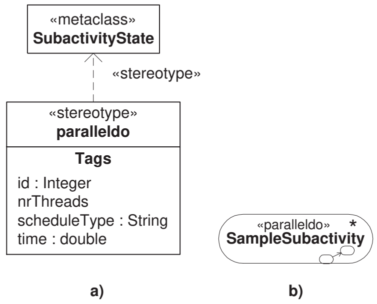


Figure 14: Modeling the PARALLEL DO Construct

4.3.4 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 15.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 Figure 15.b.

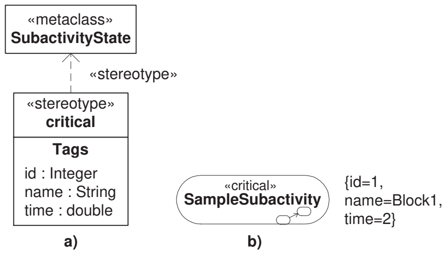


Figure 15: Modeling the CRITICAL Construct

The ORDERED construct specifies that the enclosed code is executed in the order in which iterations would be executed in a sequential execution of the loop. One thread at a time is allowed to enter an ordered section. Threads enter in the section in the order of the loop iterations. Figure 16 outlines how to model an ORDERED construct. The value of the tag *type* specifies that the stereotype *action+* represents an ordered *ActionState*.

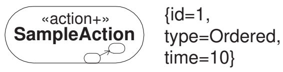


Figure 16: Modeling the ORDERED Construct

Other shared memory concepts, such as SECTIONS, are modeled analogously (Pllana and Fahringer 2002).

5 CASE STUDY: CASINO

The understanding of the nature and properties of real materials is substantially improved by theoretic study and prediction of electronic properties of atoms, molecules and solids. The CASINO (Needs et al. 2000) application provides an accurate description of the many-body physics by using Quantum Monte Carlo (QMC) methods. The QMC code carries out diffusion Monte Carlo (DMC) calculations which are computationally intensive and require high performance computing facilities in order to study realistic systems. These calculations involve a stochastic simulation where the configuration space is sampled by many points, each of which undergoes a random walk. CASINO has been encoded as a Fortran mixed OpenMP/MPI code that uses both shared memory and message passing parallelism and comprises 132 file modules.

In the following we demonstrate how to model the CASINO application by using the UML constructs introduced in this paper. Due to space limitations, we will concentrate on those application parts that are most interesting in terms of shared memory and message passing parallelism.

Figure 17.a visualizes a high-level activity diagram of the CASINO application. Each subactivity state represents a portion of the existing Fortran code. The subactivity states can be further decomposed in order to provide a hierarchical view of the entire application. Subactivities can be zoomed in and viewed as a separate more detailed activity diagram.

Figure 17.b elucidates the subactivity *Monte Carlo* in more detail. Based on values of parameters IRUN and LBUF one of the three alternative QMC methods is selected: Diffusion Monte Carlo (DMC), Diffusion Monte Carlo with buffering (DMC buf), or Variational Monte Carlo (VMC). Each of these methods is represented by a subactivity state.

The MPI parallelization of DMC employs a master-slave model, in which a master process sends work to the slave processes who complete the required work and return the results back to the master (see Figure 17.c). The numerical optimization routine runs on the master process and the set of electron configurations is divided amongst the slaves. The master process broadcasts the results to the slaves. Each of the slaves evaluates various quantities dependent on its subset of configurations. The resulting quantities are sent back to the master which determines new values of the optimization parameters. This cyclic process stops when the optimization parameter values converge.

The OpenMP PARALLELDO construct is used to parallelize two main loops of DMC, which are represented by subactivities *Initialize an ensemble of walkers* and *Explore the configuration space* in Figure 17.d. This is again a high level model, because the loop represented by the subactivity *Explore the configuration space* contains about 500 lines of code and more than 30 procedure calls. A key advantage of UML is its ability to represent applications at the desired level of abstraction and detail. The subactivity *Explore the configuration space* is represented in more detail in Figure 17.e. The OpenMP ORDERED construct is used to ensure that the configurations are copied in the same order as in the sequential version of the code, which is represented by action state *Copy the configuration*.

MPI calls are made from within serial regions of the code in order to ensure that the code is portable to systems without thread-safe MPI implementations. Inside of parallel regions the MPI calls have been placed within a CRITICAL region. By doing so, it is guaranteed that MPI calls are processed sequentially while ensuring that every thread on every process has a copy of the electron configuration data (Figure 17.f).

The CASINO application is mapped onto a process topology (see Figure 17.c) by following the Single Program Multiple Data approach which is described in (Pllana and Fahringer 2002). The activity diagram of CASINO application is associated with a single class *Process* by using *swimlanes*. An activity diagram may be divided visually into swimlanes, each separated from neighboring swimlanes by vertical solid lines on both sides. Each action is assigned to one swimlane. In our approach swimlanes are incorporated to assign the execution of an action or a subactivity to specific processes. Instances of the class *Process* are mapped onto processing units of the given process topology as shown in Figure 17.c.

6 CONCLUSIONS

The emerging of the Unified Modeling Language (UML) as the de facto standard visual modeling language which is widely used for modeling object oriented sequential applications, has unclosed new opportunities for modeling of parallel and distributed systems.

In this paper, we have described a set of UML building blocks that model some of the most important constructs of message passing and shared memory parallel paradigms which can be used to develop models for large and complex parallel and distributed applications. Moreover, UML models can be annotated with arbitrary information such as performance and control flow information which is important for a subsequent performance analysis that uses annotated UML models.

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. UML is a widely used standard and, therefore, offers a substantial advantage compared to approaches that use ad-hoc or self-defined model representations.

Currently we are in the process to develop a UML based performance estimation tool named Performance Prophet - (Fahringer and Pllana 2002). We are building a simulator that determines the performance behavior of parallel and distributed applications based on 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 without the need to encode full applications.

ACKNOWLEDGMENTS

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

REFERENCES

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

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

Fahringer, T., and S. Pllana. 2002. Performance Prophet. Institute for Software Science, University of Vienna. Available online via `<http://www.par.univie.ac.at/project/prophet>` [accessed May 15, 2002].

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

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

Needs, R., G. Rajagopal, M. Towler, P. Kent, and A. Williamson. 2000. *CASINO version 1.0 User's Manual*. Cambridge: University of Cambridge.

OMG 2001, September. Unified Modeling Language Specification. Available online via <http://www.omg.org> [accessed May 15, 2002].

OMG 2002, January. OMG XML Metadata Interchange (XMI) Specification. Available online via <http://www.omg.org> [accessed May 15, 2002].

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

Rossetti, M., B. Aylor, R. Jacoby, A. Prorock, and A. White. 2000, December 10-13. SIMFONE': An Object-Oriented Simulation Framework. In *Proceedings of the 2000 Winter Simulation Conference, J.A. Joines, R.R. Barton, K. Kang, and P.A. Fishwick, eds.* 1855–1864. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers.

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

Selic, B., and J. Rumbaugh. 1998, March. Using UML for Modeling Complex Real-Time Systems. Available online via <http://www.rational.com/products/whitepapers/100230.jsp> [accessed May 15, 2002].

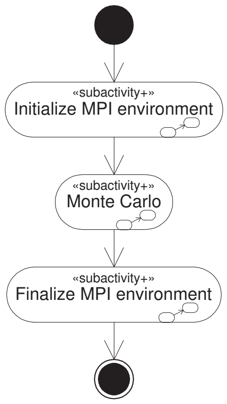
Snir, M., S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. 1996. *MPI: the complete reference*. Cambridge, MA, USA: MIT Press.

Walther, M., J. Schirmer, P. T. Flores, A. Lapp, T. Bertram, and J. Petersen. 2001. 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.

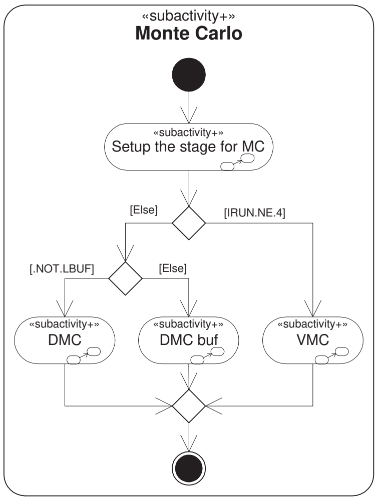
AUTHOR BIOGRAPHIES

SABRI PLLANA is a Research Assistant at the Institute for Software Science, University of Vienna. He received the Masters degree in Computer Science in 1997 from University of Zagreb, Croatia. His research focuses on performance prediction of computing systems by simulation. His email and web addresses are <pllana@par.univie.ac.at> and <www.par.univie.ac.at/~pllana>.

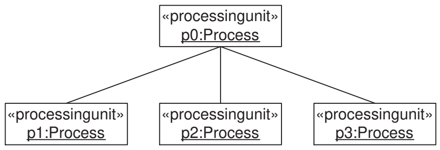
THOMAS FAHRINGER is an Associate Professor of Computer Science at the Institute for Software Science, University of Vienna. He received a Masters degree in 1988 and a Ph.D. in 1993, all in Computer Science from the Technical University of Vienna, Austria. From 1988 - 1990 he was a visiting scientist at the Engineering Design Research Center at Carnegie Mellon University in Pittsburgh, PA. From 1990 - 1998 he was Assistant Professor at the Institute for Software Technology and Parallel Systems, University of Vienna. His research focuses on programming paradigms and software tools for distributed and parallel systems. His email and web addresses are <tf@par.univie.ac.at> and <www.par.univie.ac.at/~tf>.



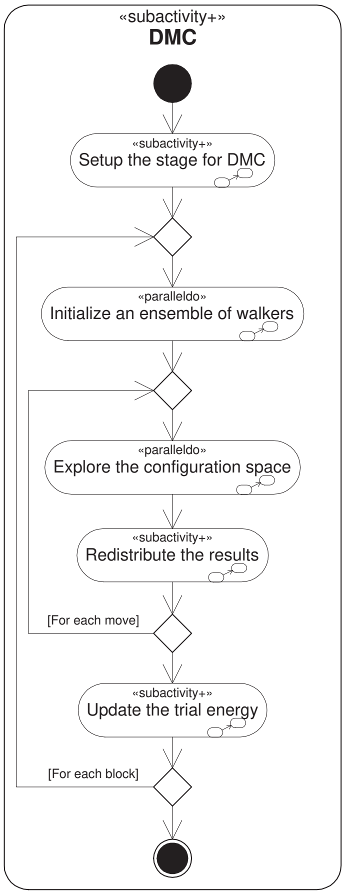
(a) High-Level UML Model of CASINO



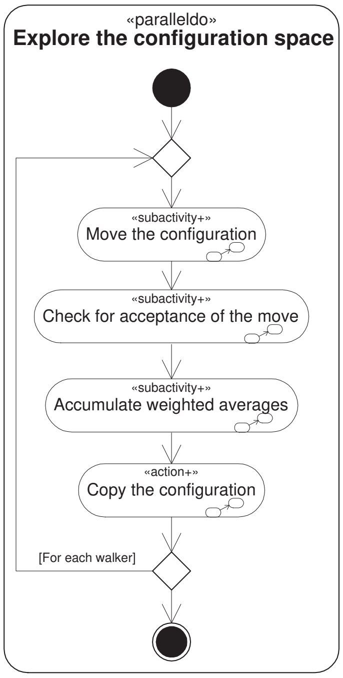
(b) Quantum Monte Carlo



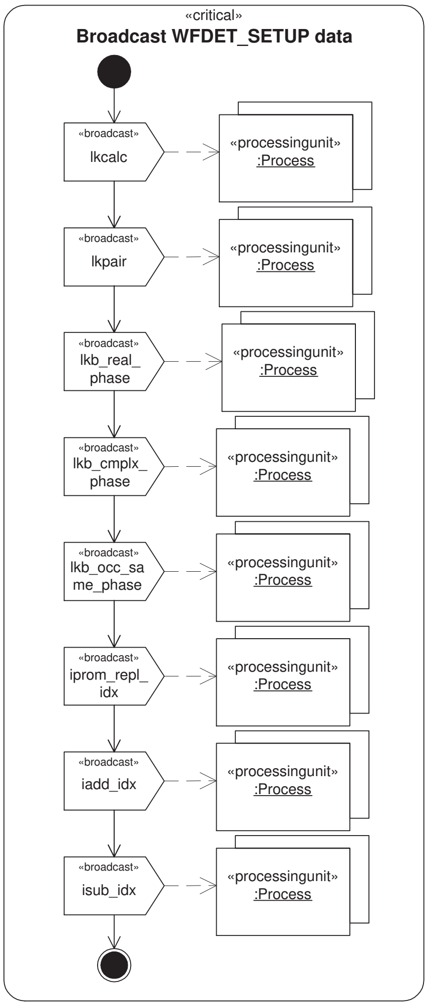
(c) Process Topology for 4 Processes



(d) Diffusion Monte Carlo



(e) Explore the Configuration Space



(f) MPI Broadcasts within a Critical Region

Figure 17: UML Model for the CASINO Application