Model-Driven Development with eUML-ARC

Kevin Marth and Shangping Ren
Illinois Institute of Technology
Department of Computer Science
Chicago, IL USA
martkev@iit.edu

ABSTRACT

Model-Driven Development (MDD) with eUML-ARC uses a synthesis of executable UML and the ARC (Agent, Role, Coordination) programming model. An entity in the ARC model is composed of concurrent role-based agents, enabling collaboration-based design and exposing both inter-entity and intra-entity parallelism, thereby facilitating the development of software systems that execute efficiently on multi-core hardware. Concurrency in eUML-ARC is based on the Actor model, which provides a simpler and more formal treatment of concurrency than found in standard UML or other approaches to executable UML. The coordination required by collaboration-based designs is separated from other computation and enacted by coordination agents upon coordinated role-based agents. In this paper, we examine the distinguishing features of the eUML-ARC approach to MDD, including the support for hierarchical state machines, the simplified concurrency model, the structure of the ARC model, and coordination viewed as an orthogonal concern. As a case study, a benchmark system is specified as an eUML-ARC model and deployed to a multi-core computer.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*; D.2.2 [**Design Tools and Techniques**]: Computer-Aided Software Engineering; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Design, Languages

Keywords

Actor Model, Coordination, Hierarchical State Machine, Executable UML

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'12 March 25-29, 2012, Riva del Garda, Italy.
Copyright 2011 ACM 978-1-4503-0857-1/12/03 ...$10.00.

1. INTRODUCTION

Multi-core processors have entered the computing mainstream, and many-core processors with 100+ cores are predicted within this decade. The absence of a clear software strategy for exploiting this increasing hardware parallelism has convinced leading computer scientists that many practicing software engineers cannot effectively program state-of-the-art processors [9]. We believe that a basis for simplifying parallel programming exists in established software technology, including the Actor model [1] and executable UML. The advent of multi-core processors has galvanized interest in the Actor model, as the Actor model has a formal foundation and provides an intuitive parallel programming model. A leading parallel research program has advocated partitioning the "software stack" into a productivity layer and an efficiency layer [8]. The efficiency layer maps a software system developed in the productivity layer to an efficient parallel implementation, and the productivity layer enables mainstream programmers to develop software systems while being shielded from the parallel hardware platform. Thus, to leverage the Actor model, software systems should be specified using a programming model that maps readily to the Actor model and its treatment of parallelism.

The separation of platform concerns is the foundation of the Model-Driven Architecture (MDA) [6]. The MDA is one approach to Model-Driven Development (MDD). When using MDD, a model of a software system is the primary development artifact throughout the lifetime of the system, from requirements analysis through deployment. A model specification is both abstract and precise, enabling the generation of other development artifacts, such as source code. The MDA is based on a platform-independent model (PIM) of the software system and the translation from a PIM to a platform-specific model (PSM) with specific choices of software platform technology and underlying hardware platform. A PSM is translated to a platform-specific implementation (PSI) for deployment as an executable.

Executable UML uses specialized profiles of the Unified Modeling Language [13] to support the MDA and enable the specification of an *executable* PIM for a software system, since executable UML requires that the PSM for a software system be generated through automated translation (not manual elaboration) of the PIM for the system. An ideal executable UML specification is an abstract PIM that includes only the precise domain-level logic required to realize a deployable software system, and an appropriate model compiler is responsible for realizing a PSI directly from the PIM. The existing approaches to executable UML have pri-

marily addressed independence from the software platform, including middleware and software architectures (e.g. client-server, 3/N-tier, peer-to-peer). The parallelism available on the hardware platform has not been a primary consideration. It is simply assumed that an executable UML model compiler will efficiently utilize the hardware platform and effectively coordinate and synchronize collaborating parallel objects, using only stereotypes and tags applied within the model (by modelers who may not be skilled in parallelism), along with low-level UML models of hardware resources.

We believe that a more effective approach to executable UML enables a model of a software system to directly expose abstract parallelism. In this paper, we present the eUML-ARC approach, which uses a synthesis of executable UML (eUML) and the ARC (Agent, Role, Coordination) programming model. An entity in the ARC model is composed of one intrinsic agent and multiple extrinsic, role-based agents. An agent is an active object whose behavior is specified in a hierarchical state machine (HSM). An agent maps directly to an actor in the Actor model. The agents that compose an entity execute concurrently, exposing both inter-entity and intra-entity parallelism, thereby facilitating the development of software systems that execute efficiently on parallel hardware. Concurrency in eUML-ARC is based on the Actor model, which provides a simpler and more formal treatment of concurrency than found in standard UML or other approaches to executable UML. Role-based agents enable collaboration-based design. The coordination required by collaborating role-based agents is separated from other computation and enacted by coordination agents.

The following sections examine the distinguishing features of eUML-ARC, including HSM support, the concurrency model, the structure of the ARC model, and coordination viewed as an orthogonal concern. As a case study, a benchmark system is specified as an eUML-ARC model and deployed to a multi-core computer. Section 2 surveys the eUML-ARC modeling language, focusing on HSM support and the concurrency model. Section 3 examines the ARC programming model. Section 4 presents an example and initial empirical evidence of the effectiveness of the ARC programming model on multi-core hardware. Section 5 concludes the paper.

2. THE MODELING LANGUAGE

2.1 Related Work

Several approaches to executable UML exist [5][7][10], and each approach enables a software system to be specified by using the following process.

- The software system is decomposed into domains.
- Each domain is modeled in a class diagram.
- Each class is modeled with structural and behavioral properties, including associations, attributes, operations, and a state machine.
- Each operation method and state machine action is implemented in a formal action language.

In xUML [5] and xtUML [10], only simple state machines are supported, and many HSM features are not available. In contrast, all standard UML HSM features are supported in eUML-ARC, with the exception of features that imply concurrency within a HSM. The foundational standard for executable UML models (fUML) [14] specifies the semantics of the UML constructs considered to be used most often. As such, the fUML specification does not address all state machine features and explicitly does not support state machine features such as change events and time events. In eUML-ARC, concurrency is based on a formal model (the Actor model), while other approaches to executable UML treat concurrency only as an operational requirement for the model compiler to synchronize the conceptual threads of control associated with active class instances.

2.2 HSM Support in eUML-ARC

HSM support in eUML-ARC exists because hierarchical states facilitate *programming by difference*, where a substate inherits behavior from superstates and defines only behavior that is specific to the substate. A design invariant can be specified once at the appropriate level in a state hierarchy, eliminating redundancy and minimizing maintenance effort. Standard UML supports a variant of Harel statecharts [3] that enables behavior to be specified using an extended HSM that combines Mealy machines, where actions are associated with state transitions, and Moore machines, where actions are associated with states. The support for HSM events, states, and transitions in eUML-ARC is described here.

2.2.1 Events

As in the Actor model, agents in eUML-ARC communicate using only asynchronous message passing. A message received by an agent is dispatched to its HSM as a *signal* - a named entity with a list of parameters. eUML-ARC supports three kinds of HSM events:

- a *signal* event that occurs when a signal is dispatched,
- a *time* event that occurs when a timer expires after a specified duration, and
- a *change* event that occurs when a Boolean expression becomes true.

Events are processed serially and to completion. Although there can be massive parallelism among agents, processing within each agent is strictly sequential.

2.2.2 States

A state in a HSM can have several features: entry actions, exit actions, transitions, deferred events, and a nested state machine. *Entry* actions and *exit* actions are executed when entering and exiting the state, respectively. *Deferred* events are queued and handled when the state machine is in another state in which the events are not deferred. A state in a state machine can be either simple or composite. A composite state has a nested state machine.

A composite state in a standard UML HSM can have multiple orthogonal regions, and each orthogonal region has a state machine. Orthogonal regions within a composite state introduce concurrency within a HSM, since the state machine within each region of a composite state is active when the composite state is active. Standard UML also supports a *do* activity for each state that executes concurrently with any *do* activity elsewhere in the current state hierarchy. The Actor model does not allow concurrency within an actor. To

align with the Actor model, eUML-ARC disallows concurrency within a HSM and does not support *do* activities or orthogonal regions. In practice, orthogonal regions are often not independent and share data. The UML standard states that orthogonal regions should interact with signals and should not explicitly interact using shared memory. Thus, replacing orthogonal regions in eUML-ARC by coordinated peer agents is appropriate.

2.2.3 Transitions

A transition in a state has several parts: an event trigger, a guard condition, a target state, and transition actions. The event trigger is an instance of one of the three kinds of events described above that initiates the transition. A triggerless transition is initiated as soon as any entry actions in the state are completed. The guard condition is an optional Boolean expression associated with the event trigger that enables the transition when the expression evaluates to true. If an event trigger occurs but its associated guard evaluates to false, the event is discarded. The target state is the state entered when the transition occurs. The transition actions are executed after the exit actions of the source state are executed and before the entry actions of the target state are executed. The inheritance of behavior in a HSM implies a distinction between the current state of the HSM and the source state that defines a transition applicable in the current state.

The UML standard defines four kinds of transitions that are all supported in eUML-ARC: internal, external, local, and start. Internal transitions are handled within a state, without causing a change in state and without causing execution of entry/exit actions for the state. External transitions always cause the exit actions of the source state to be executed and the entry actions of the target state to be executed. Local transitions are identical to external transitions except when the source and target states have a direct lineage. A local transition does not cause execution of exit actions for the source state if the target state is nested in the source state, and a local transition does not cause execution of exit/entry actions for the target state if the source state is nested in the target state. Each composite state has a start transition triggered after the execution of any entry actions for the composite state when the composite state is the target state of a transition. A start transition selects a nested substate as the next target state of the transition as it continues until a simple state is reached. A start transition can have transition actions but cannot have an event trigger or a guard condition. The following examples illustrate transitions specified in eUML-ARC.

- An external transition triggered by signal `X`, with guard `i > 0`, target state `S1`, and action `a()`:
  `event X [i > 0] (E)-> S1 { a(); }`
- A local transition triggered by signal `Y`, with no guard, target state `S2`, and no action: `event Y (L)-> S2;`
- An internal transition triggered by signal `Z`, with guard `j < 0` and actions `b();c()`:
  `event Z [j < 0] { b(); c(); }`
- A start transition with target state `S3` and no action:
  `(S)-> S3;`

2.3 Concurrency

The treatment of concurrency in standard UML is very complex. An active object (agent) has a dedicated conceptual thread of control, while a passive object does not. The calls to operations for active classes and passive classes can be either synchronous or asynchronous, and it is possible to combine operations and a state machine when defining the behavior of an active class. An active object in standard UML can be either sequential or concurrent, depending upon whether it has a state machine and whether the state machine uses operation calls as state machine events. A passive object can also be either sequential or concurrent, since each operation of a passive class is defined to be sequential, guarded, or concurrent.

The treatment of concurrency in existing executable UML approaches (including fUML) is simpler, but it is still possible to have multiple threads of control executing concurrently within an agent. The treatment of concurrency is further streamlined in eUML-ARC.

- A passive class can define only synchronous, sequential operations.
- A passive object is encapsulated within one agent, and an agent interacts with its passive objects only through synchronous operation calls.
- Agents interact only through asynchronous signals sent to state machines, where signal events are interleaved serially with other HSM events.

This treatment of agent interaction ensures that agents are internally sequential and avoids the complexities of concurrent access to the internal state of an agent. With these simplifications, eUML-ARC aligns with the Actor model and does not require the modeler to explicitly synchronize concurrent access to shared memory.

3. THE ARC PROGRAMMING MODEL

It has been argued that if concurrency was intrinsically difficult, humans would not function well in a physical world filled with concurrent activity. Of course, humans and other entities, both animate and inanimate, often successfully play several roles concurrently. The roles played by an entity that are of interest to observers determine the points-of-view from which the entity is considered when developing a software system. Consequently, the role concept has significantly influenced software development for several decades. Parallel role-based agents offer a convenient and intuitive programming model for structuring a PIM. The executable PSI generated for a PIM uses parallel role-based agents to exploit multi-core parallelism while shielding modelers from the complexities of coordination and synchronization.

Parallelism among collaborating role-based agents implies the need for coordination. Communication among agents is asynchronous and may be subject to arbitrary delay, which implies that the order in which signals are received by agents may be nondeterministic. If an agent is required to process a set of asynchronous signals in a deterministic order, such determinism must be specified within the behavior of the agent. Similarly, if a set of agents is required to process a set of signals in a deterministic order, behavior protocols that ensure the desired determinism must be specified. When such protocols are specified for individual agent behaviors,

the protocols result in cross-cutting concerns that become entangled within agent behaviors.

The ARC programming model is a novel but evolutionary synthesis of established software technologies, including role-based modeling and split objects. The ARC model has several distinguishing features.

- Entities are composed of role-based agents, exposing inter-entity and intra-entity parallelism and enabling collaboration-based designs.
- The coordination required by collaboration-based designs is separated from other computation and enacted via coordination agents upon role-based agents.
- The PSI generated from a PIM maps the model-level parallelism to the hardware platform, exploiting the available processor-level parallelism.

This section introduces entities composed of agents and organized on the basis of a synthesis of role-oriented modeling and split objects. When the roles played by an entity are largely independent, the roles may effectively execute in parallel, subject only to data dependencies within the entity. This simple observation is fundamental to the design of the ARC programming model.

3.1 Entities and Roles

The use of multi-core processors increases the importance of maximizing the parallelism within a software system, but the need to increase parallelism within a PIM must be balanced with an intuitive and convenient programming model. The ARC programming model is synthesized from two established models that are based on the notion of software viewpoints: role-oriented modeling and split objects. In role-oriented modeling [4], the features of an object are classified as either intrinsic or extrinsic. The intrinsic features of an object are allocated directly to the object, while the extrinsic features are allocated to the various roles played by the object in order to collaborate with other objects in an application. The following role properties are commonly accepted.

- Abstractivity: Role inheritance can exist.
- Aggregation: A role can be composed of other roles.
- Dependency: A role cannot exist without an object.
- Dynamicity: A role can be assumed/dropped dynamically.
- Identity: An object and its roles constitute a single entity with one identity.
- Multiplicity: An object can play several roles simultaneously, including multiple instances of the same role.
- Visibility: Object access can be restricted to one role.

In the split object model [2], a split object is a collection of parts that share a common identity. A split object denotes a single entity, and each part of a split object denotes a viewpoint of the entity. A natural synthesis of role-oriented modeling and the split object model results when the viewpoints of an entity are equated with roles, effectively partitioning an entity based on the roles it plays.

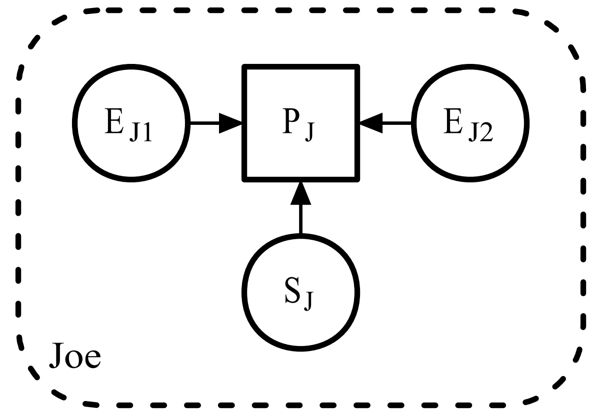

Figure 1: Person Entity with Employee and Student Roles in the ARC Model

The synthesis of role-oriented modeling and split objects is realized in the ARC model by structuring computation in terms of entities composed of `base` agents and `role` agents. A role agent is dependent on its parent agent within an entity. The parent of a role agent is either a base agent, or in the case of role aggregation, another role agent. A base agent may be created by an arbitrary agent, and no dependent parent relationship is implied. An entity is composed of one base agent that represents the intrinsic part of the entity and zero or more role agents that represent the extrinsic parts of the entity. Each part of the entity has a dedicated conceptual thread of control. Thus, the intrinsic and extrinsic parts of the entity may all execute in parallel, subject only to the data dependencies within the entity. An entity is a semantic concept and not a syntactic construct.

An example of an entity in the ARC model is illustrated in Fig. 1, where base agents are denoted by squares, role agents are denoted by circles, and solid arrows indicate parent relationships among base agents and role agents. The entity `Joe` has four agent parts. The intrinsic part of `Joe` defines features that are invariant across roles, e.g. the name of a person, and is implemented by an instance of the `Person` base agent (P_J). The example assumes that `Joe` is a student who also works two jobs. Thus, an instance of the `Student` role agent (S_J) and two instances of the `Employee` role agent (E_{J1},E_{J2}) implement the extrinsic parts of `Joe`. Role agents define features that are specific to a role or viewpoint of an entity, e.g. the phone number of a person. The role property of dynamicity is apparent in the ability to dynamically create or delete role agents that implement extrinsic parts of an entity. The role property of multiplicity is apparent in the simultaneous existence of multiple roles for `Joe`, including two `Employee` role agent parts of `Joe`.

3.2 Coordination in the ARC Model

The entities within an application do not exist in isolation. Application features require collaborations among entities. A collaboration is a collection of roles played by participating entities that cooperate to realize an application feature. In the ARC model, an entity participates in collaborations through its role agents. The parallelism among role agents within a collaboration implies the need for coordination if the role agents are to cooperate effectively. Coordination in the ARC model is enacted by **coordinator** agents and

is based on open and dynamic sets of role agents. Both coordinator agents and role agents may behave as **coordination** agents and react to events that occur within other observed agents. To enhance simplicity and locality, a role agent observes only its parent agent, and a coordinator agent observes only the current elements of its coordinated set(s) of role agents.

A coordinator agent is the point of contact for an instance of a feature collaboration and also enacts coordination in reaction to events observed in coordinated role agents. The HSM that specifies the behavior of a coordinator agent therefore handles signals from clients of a feature collaboration as well as signals from coordinated role agents collaborating within the feature. The signals sent to a coordinator from coordinated role agents typically signal when a coordinated role agent enters or exits a specified state. The coordination agents in the ARC model enhance the modularity of features. A new application feature typically requires new role agents, new operations for the respective parent agents to instantiate the new role agents, and a new coordinator agent to coordinate the role-based collaboration within the feature. The existing role agents of entities often remain oblivious to additional role agents.

4. AN eUML-ARC EXAMPLE

We illustrate the eUML-ARC model by solving a version of the dining philosophers problem. Each philosopher alternates between thinking and dining. While thinking, each philosopher contributes to solving the NAS Embarrassingly Parallel (EP) benchmark [12]. The EP benchmark is typical of Monte-Carlo simulation applications in scientific computing and provides a concrete representation of a workload that is easily partitioned among ARC parallel agents.

4.1 The Dining Philosophers in eUML-ARC

The eUML-ARC solution to the dining philosophers builds on the example entity illustrated in Fig. 1. In addition to the `Employee` and `Student` roles, each `Person` also plays the `Philosopher` role in a collaboration of dining philosophers. A `DP` coordinator agent coordinates the `Philosopher` role agents by behaving as a simple barrier to separate successive phases of the EP benchmark solution. In even phases of the solution, philosophers assigned an even index number are in the thinking state and contribute to the EP benchmark solution, while philosophers assigned an odd index number are in the dining state. In odd phases of the solution, the states are reversed. Adjacent philosophers are always in different states and can therefore atomically acquire necessary tableware (fork, chopstick) without the possibility of deadlock. Before considering the required coordination, we summarize the behavior of `Philosopher` agent specified in Fig. 2.

- In the `start` state, the philosopher transitions to the `thinking` state or the `dining` state, based on whether its index number is even or odd, respectively.
- On input of the `run` signal in the `thinking` state, the philosopher executes a step of the benchmark via the `EP` procedure and then transitions to the `dining` state.
- On input of the `run` signal in the `dining` state, the philosopher executes the `eat` procedure and then returns to the `thinking` state.

```
agent {role} Philosopher
{
  agent (index:Integer, steps:Integer) {
    this.index := index; this.steps := steps;
  }
  index:Integer; steps:Integer;
  (S)-> (index mod 2) == 0 ? thinking : dining;
  state thinking {
    event run (E)-> dining
    { EP(index, run.step, steps); }
    exit { coordinator.end(index); }
  }
  state dining {
    event run (E)-> thinking { eat(); }
    exit { coordinator.end(index); }
} }

agent {coordinator} DP
{
  agent (p:Philosopher[*], steps:Integer) {
    this.philosophers := p; this.steps := steps;
  }
  step,steps:Integer; await:Integer;
  philosophers:Philosopher[*];
  send(step:Integer):void {
    for (philosopher:Philosopher in philosophers)
      philosopher.run(step);
    await := philosophers.size();
  }
  (S)-> barrier { send(step := 1); }
  state barrier {
    input end {
      if ((await -= 1) > 0); else
      if ((steps -= 1) > 0)  send(step += 1); else
      stop;
} } }

signal run(step:Integer);
signal end(index:Integer);
```

Figure 2: The Dining Philosophers Problem

The coordination enacted in Fig. 2 by the `DP` coordinator is triggered by signals sent from the `Philosopher` role agent. Coordination in the ARC model is treated as an orthogonal concern, and coordination logic is typically implemented in the `entry` and `exit` actions for states, since state transitions are fundamental coordination events. The following coordination logic is specified in Fig. 2.

- When a `Philosopher` exits the `thinking` state, the `end` signal is sent to its `DP` coordinator.
- When a `Philosopher` exits the `dining` state, the `end` signal is sent to its `DP` coordinator.

The EP benchmark can be partitioned into multiple computational steps of similar granularity, and the `DP` coordinator is parameterized by the number of steps to use when solving the benchmark. Despite the roughly equal workloads shared by the philosophers in each step, there is no guarantee that individual philosophers will finish a step and advance to the next step in a synchronized fashion without explicit coordination. The barrier implemented by the `DP` coordinator ensures that each step is synchronized. The `DP` coordinator initiates a step by sending the `run` signal to each philosopher, and the `DP` coordinator then waits for the `end` signal from each philosopher before initiating the next step. The barrier-based coordination ensures that adjacent philosophers are always in different states and eliminates the explicit locking typically used to achieve mutual exclusion in the dining philosophers problem.

Execution Time (seconds) and Parallel Efficiency (%)					
Input Size	NAS serial	phil=2,cores=1	phil=4,cores=2	phil=8,cores=4	phil=16,cores=8
2^{28}	22.75s	22.75s	11.50s	5.75s	3.00s
2^{28}	-	100.00%	98.91%	98.91%	94.79%
2^{30}	90.00s	90.00s	45.75s	22.75s	11.75s
2^{30}	-	100.00%	98.36%	98.90%	95.74%
2^{32}	360.00s	360.00s	181.00s	93.00s	46.50s
2^{32}	-	100.00%	99.45%	96.77%	96.77%

Figure 3: Performance of the eUML-ARC Dining Philosopher PSI on an Intel Xeon X5570

4.2 Performance Evaluation

The table in Fig. 3 summarizes the performance achieved when comparing the eUML-ARC PSI of the dining philosophers EP benchmark to the NAS serial version of the EP benchmark implemented in C. The target platform was the Intel Xeon model X5570, a two-processor machine with four cores per processor, providing eight total hardware threads, each running at 2.93 GHz.

The EP benchmark is called "Embarrassingly Parallel" because it is compute-bound, and ideal implementations of the EP benchmark demonstrate linear speedup and 100% parallel efficiency as the benchmark is partitioned among additional cores. The parallel efficiency measures how well the PSI utilizes the multi-core parallelism, including the overhead of communication and coordination. Efficiency is defined by the formula $E = T_1/(T_t * t)$, where T_1 is the execution time of the NAS serial version, and T_t is the execution time of the eUML-ARC parallel version using t threads and $(t*2)$ philosophers. The PSI matches the number of application threads to the available hardware threads. The conceptual threads dedicated to parallel agents are mapped onto the application threads by the PSI. An agent is scheduled in an available application thread when a signal is dispatched to the agent, and the agent is run to completion on the input signal in the assigned thread.

The performance achieved by the eUML-ARC PSI of the dining philosophers EP benchmark demonstrates that only modest overhead is inherent with communicating hierarchical state machines and role-based coordination. The parallel efficiency of the eUML-ARC model with an arbitrary application obviously depends upon how effectively exploitable parallelism can be exposed by the role-based ARC model and the effective use of role-based coordination.

5. CONCLUSION

The use of multi-core processors is reshaping the development of software applications. The eUML-ARC model aligns with the principles of executable UML and enables abstract platform-independent parallelism. Aligning the Actor model and executable UML in eUML-ARC provides a concurrency model that exploits inter-agent parallelism while ensuring that agent behaviors retain the familiarity and simplicity of sequential programming. The eUML-ARC model uses a synthesis of role-oriented modeling and the split object model to structure computation in terms of entities. An entity is composed of a base intrinsic agent and multiple extrinsic role agents, all with dedicated conceptual threads of control. Entities interact through their role agents in the context of feature-oriented collaborations with dedicated coordinator agents that enact coordination upon the role agents.

The initial experience gained by specifying a small number of benchmark applications in the eUML-ARC model has been positive. We plan to specify and measure additional benchmark software systems. The effective utilization of representative multi-core architectures can then be more fully assessed. We believe the resulting evidence will position the eUML-ARC model as an attractive technology for the development of software systems that execute efficiently on parallel multi-core hardware.

6. REFERENCES

[1] Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press (1986)

[2] Bardou, D., Dony, C.: Split Objects: A Disciplined Use of Delegation Within Objects. ACM SIGPLAN Notices, 31(10) 122–137 (1996)

[3] Harel, D.: Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming, 8(3) 231–274 (1987)

[4] Kristensen, B.B.: Object-Oriented Modeling with Roles. Proceedings of the 2nd International Conference on Object-Oriented Information Systems, 57–71 (1996)

[5] Mellor, S.J., Balcer, S.J.: Executable UML: A Foundation for Model-Driven Architecture. Addison Wesley (2002)

[6] Mellor, S.J., Kendall, S., Uhl, A., Weise, D.: MDA Distilled. Addison Wesley (2004)

[7] Milicev, D.: Model-Driven Development with Executable UML. Wiley (2009)

[8] Patterson, D. et al.: A View of the Parallel Computing Landscape. Communications of the ACM, 52(10) 56–67 (2009)

[9] Patterson, D.: The Trouble with Multi-Core. IEEE Spectrum, 47(7) 28–32 (2010)

[10] Raistrick, C., Francis, P., Wright, J., Carter, C., Wilkie, I.: Model Driven Architecture with Executable UML. Cambridge University Press (2004)

[11] Samek, M.: Practical UML Statecharts in C/C++. Elsevier (2009)

[12] The NAS Parallel Benchmarks. www.nas.nasa.gov/Resources/Software/npb.html

[13] Object Management Group: UML Superstructure Specification, Version 2.1.2. www.omg.org/docs/formal/07-11-02.pdf

[14] Object Management Group: Semantics of a Foundational Subset for Executable UML Models (fUML), Version 1.0. www.omg.org/spec/FUML