Actor-eUML for Concurrent Programming

Kevin Marth and Shangping Ren

Illinois Institute of Technology
Department of Computer Science
Chicago, IL USA
martkev@iit.edu

Abstract. The advent of multi-core processors offers an opportunity to increase the usage of Executable UML. Researchers are advocating the division of software systems into a productivity layer and an efficiency layer to shield mainstream programmers from the complexities of parallelism. Such separation of application and platform concerns is the foundation of Executable UML. To leverage this opportunity, an approach to Executable UML must address the complexity of the UML standard and provide a formal model of concurrency. In this paper, we introduce the Actor-eUML model and formalize the mapping between actors in the Actor model and Executable UML agents (active objects) by unifying the semantics of actor behavior and the hierarchical state machine (HSM) semantics of Executable UML agents. The UML treatment of concurrency is simplified, and the Actor model is extended to enable a set of actor behaviors to specify the HSM for an Executable UML active class.

1 Introduction

Multi-core processors have entered the computing mainstream, and many-core processors with 100+ cores are predicted within this decade. The increasing hardware parallelism and the absence of a clear software strategy for exploiting this parallelism have convinced leading computer scientists that many practicing software engineers cannot effectively program state-of-the-art processors [8]. 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 sound formal foundation and provides an intuitive parallel programming model. To leverage the Actor model, software systems should be specified in a language that provides first-class support for the Actor model and exposes its rather abstract treatment of parallelism. A leading parallel research program has advocated dividing the "software stack" into a productivity layer and an efficiency layer [7]. Parallel concerns are addressed in the efficiency layer by expert parallel programmers, and the productivity layer enables mainstream programmers to develop applications while being shielded from the parallel hardware platform. This separation of application concerns (productivity layer) and platform concerns (efficiency layer) is the foundation of Executable UML.

Fortunately, the Actor model and Executable UML are readily unified. In this paper, we introduce the Actor-eUML model and formalize the mapping between actors in the Actor model and agents (active objects) in Executable UML by unifying the semantics of actor behavior and the hierarchical state machine (HSM) semantics of Executable UML agents. Simply stated, an Executable UML agent is an actor whose behavior is specified as a HSM. To facilitate the definition of unified semantics for Actor-eUML, we simplify the UML treatment of concurrency and extend the Actor model to enable a set of actor behaviors to specify the HSM for an Executable UML active class. Section 2 presents an overview of the Actor-eUML model. Section 3 presents the operational semantics of the Actor-eUML model. Section 4 concludes the paper.

2 Overview of the Actor-eUML Model

2.1 Related Work

The separation of application and platform concerns is embodied in the Model-Driven Architecture (MDA) [5]. Executable UML uses profiles of the Unified Modeling Language [11] to support the MDA and enable the specification of an *executable* platform-independent model (PIM) of a software system that can be translated to a platform-specific implementation (PSI) using a model compiler. Several approaches to Executable UML exist [4][6][9], and each approach enables a software system to be specified using the following process.

- The software system is decomposed into domains (concerns).
- Each domain is modeled in a class diagram using several classes.
- Each class has structural and behavioral properties, including associations, attributes, operations, and a state machine.
- A formal action language is used to specify the implementation of operation methods and state machine actions.

In both xUML [4] and xtUML [9], only simple state machines are supported, and many HSM features are not available. In contrast, all standard UML HSM features are supported in Actor-eUML, with the exception of features that imply concurrency within a HSM. The foundational subset for Executable UML models (fUML) [12] precisely 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 call events, change events, and time events.

The Actor-eUML model has a formal concurrency model (the Actor model), while existing approaches to Executable UML lack a formal treatment of concurrency beyond the operational requirement for the modeler and/or the model compiler to synchronize the conceptual threads of control associated with active class instances. Some HSM features, such as deferring certain messages when an actor is in a given state, have been implemented in actor-based programming languages using reflective mechanisms that modify the behavior of the mail queue for an actor [10], but these actor-based languages lack full-featured HSM support.

2.2 Hierarchical State Machines in Actor-eUML

The Actor-eUML model promotes HSM usage because state-based behavior is fundamental to object-based programming. 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. Actor-eUML supports internal, external, local, and start transitions as specified in standard UML and provides the following support for events and states.

Events. As in the Actor model, agents in Actor-eUML 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. Actor-eUML 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.

States. A state in an Actor-eUML HSM has several features: an optional name, 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.

In standard UML, a composite state can have multiple orthogonal regions, and each 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 avoids concurrency within an actor. To align with the Actor model, Actor-eUML does not allow concurrency within a HSM and consequently 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 Actor-eUML by coordinated peer agents is appropriate.

2.3 Simplified Concurrency in Actor-eUML

To align the Actor-eUML model with the Actor model, it is necessary to eliminate HSM features that introduce concurrency within an active object, but additional simplification is required to complete the alignment. The treatment of concurrency in standard UML is heterogeneous and complex. An active object (i.e. 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 internally sequential or internally concurrent, depending upon whether it has a state machine and whether the state machine uses operation calls as state machine triggers. A passive object can also be either internally sequential or internally concurrent, since each operation of a passive class is defined to be sequential, guarded, or concurrent.

It is apparent that the multiple interacting characteristics of the features of active and passive classes add complexity to the treatment of concurrency in standard UML. 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. Actor-eUML further streamlines the treatment of concurrency.

- 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.
- An active class can define operations, but a call to an agent operation is simply notation for an implicit synchronous signal exchange with the agent.

A call to an agent operation sends a signal with the same signature to the HSM for the agent and blocks the caller until the signal is processed and a reply signal is received. Thus, any communication with an agent is a signal event that is interleaved serially with other HSM events in the thread of control for the agent. 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, the Actor-eUML concurrency model aligns with the Actor model and is safer than multi-core programming models that require the programmer to explicitly synchronize concurrent access to shared memory.

2.4 Actors in Actor-eUML

The Actor model [1] is a formal theory of computation and concurrency based on active, autonomous, encapsulated objects that communicate exclusively through asynchronous message passing. The Actor model has a sound mathematical foundation but is also influenced by implementation concerns and the laws of physics. The Actor model acknowledges that messages can encounter bounded but indeterminate delays in transmission and can therefore be delivered out of order.

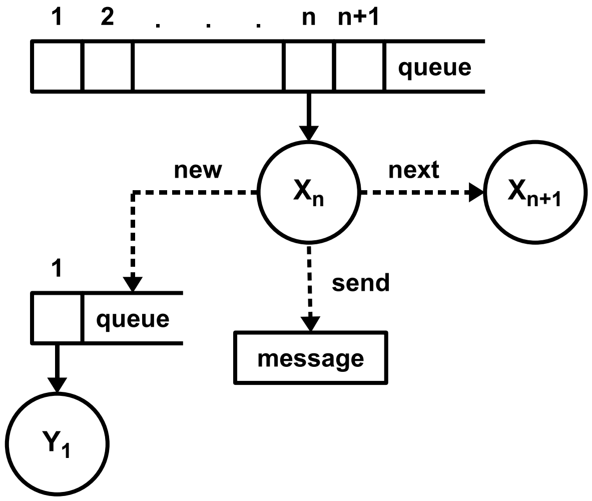

Fig. 1. An Actor in the Actor Model [1]

As illustrated in Fig. 1, in response to each message received from its abstract mailbox (external queue), an actor `X` can:

- create a finite number of new actors,
- send a finite number of messages to other actors, and
- select the behavior used to process the next message.

The Actor model is characterized by inherent concurrency among actors. An actor is allowed to pipeline the processing of messages by selecting the behavior used to process the next message and actually dispatching the next message for processing before the processing of the current message has completed. However, pipelined actor behaviors cannot share internal state, and the Actor model does not require message pipelining. The ability to pipeline messages is not compatible with HSM semantics, as the exit and entry actions for a transition must execute before the next transition can be triggered, so actors in the Actor-eUML model that realize HSM behavior do not attempt message pipelining. However, other actors in the Actor-eUML model can use message pipelining.

An actor can send messages to its own mailbox, but a message an actor sends to itself is interleaved with messages received from other actors and is not guaranteed to be the next message dispatched. This consideration and the requirement that an actor consume a message with each behavior change lead to a continuation-passing style that uses cooperating auxiliary actors to process a single client message. This style of programming adds conceptual overhead for programmers who find it confusing and adds implementation overhead that cannot always be eliminated by smart compilers and sophisticated schemes aimed at minimizing the performance impact of actor creation and communication.

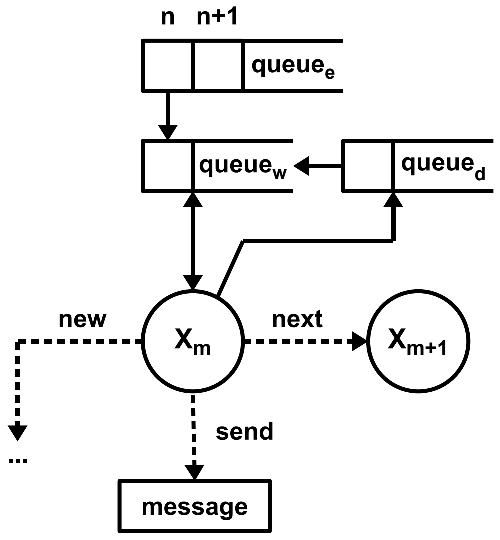

Fig. 2. An Actor in the Actor-eUML Model

The Actor-eUML model retains the essence of the pure Actor model while adding capabilities that simplify actor programming and enable HSM behavior to be expressed directly and conveniently. As illustrated in Fig. 2, the actor interface to its abstract mailbox has been extended with two internal queues: a working queue ($\texttt{queue}_\texttt{w}$) for internal messages used while processing a single external message, and a defer queue ($\texttt{queue}_\texttt{d}$) used to defer messages based on the current state in a HSM. The external queue ($\texttt{queue}_\texttt{e}$) that receives messages sent to an actor has been retained. When a message is dispatched from $\texttt{queue}_\texttt{e}$, the message is moved to $\texttt{queue}_\texttt{w}$ and then dispatched for processing by the next behavior. As the behavior executes, messages can be added to $\texttt{queue}_\texttt{w}$. When the behavior completes, the message at the head of $\texttt{queue}_\texttt{w}$ is dispatched to the next behavior. If the $\texttt{queue}_\texttt{w}$ is empty when a behavior completes, the next message is dispatched from $\texttt{queue}_\texttt{e}$. A message dispatched from $\texttt{queue}_\texttt{w}$ can be deferred to $\texttt{queue}_\texttt{d}$. The messages in $\texttt{queue}_\texttt{d}$ are moved to $\texttt{queue}_\texttt{w}$ to revisit them.

The additional internal queues enable a single actor to completely process a client message without creating and communicating with auxiliary actors and also facilitate the expression of HSM behavior. Each state in a HSM is mapped to an actor behavior, and a signal is delegated from the current state to its parent state by adding the signal to $\texttt{queue}_\texttt{w}$ and then selecting the behavior for its parent state as the next behavior. A sequence of start, entry, and exit actions is executed during a state transition by adding specialized messages to $\texttt{queue}_\texttt{w}$ and then selecting the behavior of the next state in the sequence. A signal that is deferred in the current state is added to $\texttt{queue}_\texttt{d}$. Deferred messages are revisited after a state transition by moving the messages from $\texttt{queue}_\texttt{d}$ to $\texttt{queue}_\texttt{w}$.

3 Actor-eUML Semantics

The Actor-eUML model defines the following actor primitives, where $\mathbb{B}$ is a behavior, $\mathbb{M}$ is a message, and $\mathbb{V}$ is a list of parameter values. The call $\mathbb{B}(\mathbb{V})$ returns a closure, which is a function and a referencing environment for the nonlocal names in the function that binds the nonlocal names to the corresponding variables in scope at the time the closure is created. The closure returned by the call $\mathbb{B}(\mathbb{V})$ captures the variables in $\mathbb{V}$, and the closure expects to be passed $\mathbb{M}$ as a parameter when called subsequently.

- `actor-new`$(\mathbb{B}, \mathbb{V})$: create a new actor with initial behavior $\mathbb{B}(\mathbb{V})$.
- `actor-next`$(A_i, \mathbb{B}, \mathbb{V})$: select $\mathbb{B}(\mathbb{V})$ as the next behavior for actor A_i.
- `actor-send`$(A_i, \mathbb{M})$: send $\mathbb{M}$ to the tail of $\texttt{queue}_{\texttt{e}}$ for actor A_i.
- `actor-push`$(A_i, \mathbb{M})$: push $\mathbb{M}$ at the head of $\texttt{queue}_{\texttt{w}}$ for actor A_i.
- `actor-push-defer`$(A_i, \mathbb{M})$: push $\mathbb{M}$ at the head of $\texttt{queue}_{\texttt{d}}$ for actor A_i.
- `actor-move-defer`(A_i): move all messages in $\texttt{queue}_{\texttt{d}}$ to $\texttt{queue}_{\texttt{w}}$ for actor A_i.

The {`actor-new`, `actor-next`, `actor-send`} primitives are inherited from the Actor model, and the {`actor-push`, `actor-push-defer`, `actor-move-defer`} primitives are extensions to the Actor model introduced by the Actor-eUML model. When transforming a PIM to a PSI, a model compiler for an Actor-eUML implementation translates the HSM associated with each active class to a target programming language in which the actor primitives have been embedded.

At any point in a computation, an actor is either quiescent or actively processing a message. The term $\texttt{actor}_4(A_i, Q, C, M)$ denotes a quiescent actor, where A_i uniquely identifies the actor, Q is the queue for the actor, C is the closure used to process the next message dispatched by the actor, and M is the local memory for the actor. The queue Q is a 3-tuple $\langle Q_e, Q_w, Q_d \rangle$, where Q_e is the external queue where messages sent to the actor are received, Q_w is the work queue used when processing a message $\mathbb{M}$ dispatched by the actor, and Q_d is used to queue messages deferred after dispatch. At points in a computation, a component of Q can be empty and is denoted by $Q_\perp$. The term $\texttt{actor}_5(A_i, Q, C_\perp, M, E \vdash S)$ denotes an active actor and extends the $\texttt{actor}_4$ term to represent a computation in which statement list S is executing in environment E. An active actor has a null C, denoted by $C_\perp$.

A transition relation between actor configurations is used to define the Actor-eUML operational semantics, as in [2]. A configuration in an actor computation consists of $\texttt{actor}_4$, $\texttt{actor}_5$, and `send` terms. The $\texttt{send}(A_i, \mathbb{M})$ term denotes a message $\mathbb{M}$ sent to actor A_i that is in transit and not yet received. The specification of structural operational semantics for Actor-eUML uses rewrite rules to define computation as a sequence of transitions among actor configurations.

Rules (1) and (2) define the semantics of message receipt. A message $\mathbb{M}$ sent to actor A_i can be received and appended to the external queue for actor A_i when the actor is quiescent (1) or active (2). The `send` term is consumed and eliminated by the rewrite. The message receipt rules illustrate several properties explicit in the Actor model. An actor message is an asynchronous, reliable, point-to-point communication between two actors. The semantics of message receipt are independent of the message sender. Each message that is sent is ultimately received, although there is no guarantee of the order in which messages are received. A message cannot be broadcast and is received by exactly one actor.

$$\begin{aligned} \texttt{send}(A_i, \mathbb{M})\ \ \texttt{actor}_4(A_i, \langle Q_e, Q_w, Q_d\rangle, C, M) \\ \longrightarrow\ \texttt{actor}_4(A_i, \langle Q_e{:}\mathbb{M}, Q_w, Q_d\rangle, C, M) \end{aligned} \tag{1}$$

$$\begin{aligned} \texttt{send}(A_i, \mathbb{M})\ \ \texttt{actor}_5(A_i, \langle Q_e, Q_w, Q_d\rangle, C_\perp, M, E \vdash S) \\ \longrightarrow\ \texttt{actor}_5(A_i, \langle Q_e{:}\mathbb{M}, Q_w, Q_d\rangle, C_\perp, M, E \vdash S) \end{aligned} \tag{2}$$

Rules (3) and (4) define the semantics of message dispatch. In rule (3), the quiescent actor A_i with non-empty Q_e and empty Q_w initiates the dispatch of the message $\mathbb{M}$ at the head of Q_e by moving $\mathbb{M}$ to Q_w. In rule (4), the quiescent actor A_i completes message dispatch from Q_w and becomes an active actor by calling the closure C to process $\mathbb{M}$ in the initial environment E_c associated with C. The message dispatch rules enforce the serial, run-to-completion processing of messages and the demand-driven relationship between Q_e and Q_w in the Actor-eUML model. An actor cannot process multiple messages concurrently, and a message is dispatched from Q_e only when Q_w is empty.

$$\texttt{actor}_4(A_i, \langle \mathbb{M}{:}Q_e, Q_\perp, Q_d\rangle, C, M) \longrightarrow \texttt{actor}_4(A_i, \langle Q_e, \mathbb{M}, Q_d\rangle, C, M) \tag{3}$$

$$\begin{aligned} \texttt{actor}_4(A_i, \langle Q_e, \mathbb{M}{:}Q_w, Q_d\rangle, C, M) \\ \longrightarrow\ \texttt{actor}_5(A_i, \langle Q_e, Q_w, Q_d\rangle, C_\perp, M, E_C \vdash (\texttt{call}\ C\ \mathbb{M})) \end{aligned} \tag{4}$$

Rules (5), (6), and (7) define the semantics of the `actor-new`, `actor-next`, and `actor-send` primitives, respectively. In rule (5), the active actor A_i executes the `actor-new` primitive to augment the configuration with an $\texttt{actor}_4$ term that denotes a new actor A_n with empty Q, uninitialized local memory ($M_\perp$), and initial behavior closure $C = \mathbb{B}(\mathbb{V})$. In rule (6), the active actor A_i executes the `actor-next` primitive to select its next behavior closure $C = \mathbb{B}(\mathbb{V})$ and becomes a quiescent actor. In rule (7), the active actor A_i executes the `actor-send` primitive to send the message $\mathbb{M}$ to actor A_j, where both $i = j$ and $i \neq j$ are well-defined.

$$\begin{aligned} \texttt{actor}_5(A_i, Q, C_\perp, M, E \vdash \texttt{actor-new}(\mathbb{B}, \mathbb{V}); S) \\ \longrightarrow\ \texttt{actor}_5(A_i, Q, C_\perp, M, E \vdash S)\ \ \texttt{actor}_4(A_n, \langle Q_\perp, Q_\perp, Q_\perp\rangle, C, M_\perp) \end{aligned} \tag{5}$$

$$\begin{aligned} \texttt{actor}_5(A_i, Q, C_\perp, M, E \vdash \texttt{actor-next}(A_i, \mathbb{B}, \mathbb{V}); S) \\ \longrightarrow\ \texttt{actor}_4(A_i, Q, C, M) \end{aligned} \tag{6}$$

$$\begin{aligned} \texttt{actor}_5(A_i, Q, C_\perp, M, E \vdash \texttt{actor-send}(A_j, \mathbb{M}); S) \\ \longrightarrow\ \texttt{actor}_5(A_i, Q, C_\perp, M, E \vdash S)\ \ \texttt{send}(A_j, \mathbb{M}) \end{aligned} \tag{7}$$

Rules (8), (9), and (10) define the semantics of the `actor-push`, `actor-push-defer`, and `actor-move-defer` primitives, respectively.

$$\begin{aligned} &\texttt{actor}_5(A_i, \langle Q_e, Q_w, Q_d\rangle, C_\bot, M, E \vdash \texttt{actor-push}(A_i, \mathbb{M}); S) \\ \longrightarrow\ &\texttt{actor}_5(A_i, \langle Q_e, \mathbb{M}{:}Q_w, Q_d\rangle, C_\bot, M, E \vdash S) \end{aligned} \tag{8}$$

$$\begin{aligned} &\texttt{actor}_5(A_i, \langle Q_e, Q_w, Q_d\rangle, C_\bot, M, E \vdash \texttt{actor-push-defer}(A_i, \mathbb{M}); S) \\ \longrightarrow\ &\texttt{actor}_5(A_i, \langle Q_e, Q_w, \mathbb{M}{:}Q_d\rangle, C_\bot, M, E \vdash S) \end{aligned} \tag{9}$$

$$\begin{aligned} &\texttt{actor}_5(A_i, \langle Q_e, Q_w, Q_d\rangle, C_\bot, M, E \vdash \texttt{actor-move-defer}(A_i); S) \\ \longrightarrow\ &\texttt{actor}_5(A_i, \langle Q_e, Q_w{:}\text{reverse}(Q_d), Q_\bot\rangle, C_\bot, M, E \vdash S) \end{aligned} \tag{10}$$

The actor primitives intrinsic to the Actor-eUML model are the foundation for other abstractions useful in realizing an implementation of HSM behavior. The following HSM abstractions are typically defined as macros in the target programming language. The HSM abstraction macros implement HSM behavior using exclusively the Actor-eUML primitives `actor-push` and `actor-next` and the internal work queue Q_w, in combination with parameter passing between actor behaviors. The HSM abstraction macros facilitate a direct translation from a HSM specification to its implementation.

- The `HSM-state-start` macro realizes the start transition for a state.
- The `HSM-state-entry` macro realizes the entry actions for a state.
- The `HSM-state-exit` macro realizes the exit actions for a state.
- The `HSM-state-reset` macro returns the HSM to its current state prior to delegating a signal up the HSM hierarchy without consuming the signal.
- The `HSM-state-next` macro delegates a signal to a superstate.
- The `HSM-state-transition` macro realizes a local or external transition from a source state to a target state.
- The `HSM-state-transition-internal` macro realizes an internal transition in a state.

The `actor-push-defer` and `actor-move-defer` primitives implement deferred signals within a HSM, and the `actor-send` primitive is used to send an asynchronous signal from the HSM for an agent to the HSM for a target agent.

A reference implementation of the Actor-eUML model and the associated HSM abstraction macros has been developed in Common Lisp, confirming that a direct and efficient realization of the model is practical. A C++ implementation of the Actor-eUML model is also in development and will be the target language for a model compiler, enabling software engineers to specify Executable UML models oriented to the problem space that abstract the programming details of the Actor-eUML model in the solution space. However, the mapping from the UML agents in the problem space to the Actor-eUML actors in the solution space is direct, reducing the semantic distance between a UML specification and its realization and ensuring that UML specifications are founded on a formal model of concurrency that provides a logically sound and intuitive basis for reasoning about parallel behavior and analyzing run-time performance.

4 Summary and Conclusion

The advent of multi-core processors signaled a revolution in computer hardware. We believe that it is possible to program multi-core and many-core processors by using an evolutionary approach that leverages established software technology, notably the Actor model and Executable UML. The Actor model has a sound formal foundation and provides an intuitive and safe concurrent programming model. Executable UML consolidates and standardizes several decades of experience with object-based programming. Unifying the Actor model and Executable UML in the Actor-eUML model provides a concurrency model that exploits massive inter-agent parallelism while ensuring that agent behaviors retain the familiarity and simplicity of sequential programming. The HSM is the foundation of agent behavior in Actor-eUML. The Actor-eUML model streamlines the UML concurrency model, eliminates HSM features that imply intra-agent concurrency, and introduces conservative extensions to the structure and behavior of message dispatch in the Actor model. A definition of the operational semantics of the actor primitives provided by the Actor-eUML model was presented, and a reference implementation of the Actor-eUML model is available.

References

1. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge (1986)
2. Agha, G., Mason, I.A., Smith, S.F., Talcott, C.L.: A Foundation for Actor Computation. Journal of Functional Programming, 1–72 (1997)
3. Harel, D.: Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming 8(3), 231–274 (1987)
4. Mellor, S.J., Balcer, S.J.: Executable UML: A Foundation for Model-Driven Architecture. Addison-Wesley, Reading (2002)
5. Mellor, S.J., Kendall, S., Uhl, A., Weise, D.: MDA Distilled. Addison-Wesley, Reading (2004)
6. Milicev, D.: Model-Driven Development with Executable UML. Wiley, Chichester (2009)
7. Patterson, D., et al.: A View of the Parallel Computing Landscape. Communications of the ACM 52(10), 56–67 (2009)
8. Patterson, D.: The Trouble with Multi-Core. IEEE Spectrum 47(7), 28–32 (2010)
9. Raistrick, C., Francis, P., Wright, J., Carter, C., Wilkie, I.: Model Driven Architecture with Executable UML. Cambridge University Press, Cambridge (2004)
10. Tomlinson, C., Singh, V.: Inheritance and Synchronization with Enabled Sets. SIGPLAN Notices 24(10), 103–112 (1989)
11. Object Management Group: UML Superstructure Specification, Version 2.1.2, `http://www.omg.org/docs/formal/07-11-02.pdf`
12. Object Management Group: Semantics of a Foundational Subset for Executable UML Models (fUML), Version 1.0, `http://www.omg.org/spec/FUML`