CoDE - A Software Framework for Agent-based Simulation

AGOSTINO POGGI
Dipartimento di Ingegneria dell’Informazione
Università degli Studi di Parma
Viale G.P. Usberti 181A, Parma, 43100
ITALY
agostino.poggi@unipr.it http://www.ce.unipr.it/people/poggi

Abstract: - The issue of developing scalable, concurrent and distributed programs has generated a renewed interest in message-passing languages and frameworks. CoDE is an actor-based software framework aimed at both simplifying the development of large and distributed complex systems and guarantying an efficient execution of applications. This software framework takes advantage of a concise actor model that makes easy the development of the actor code by delegating the management of events (i.e., the reception of messages) to the execution environment. Moreover, it provides a flexible and configurable implementation that allows the development of scalable and efficient applications through the possibility of using different implementations of the actor model. In this paper, we introduce the CoDE software framework and discuss its use for agent-based simulation.

Key-Words: - Actor model, software framework, agent-based simulation, distributed system, Java.

1 Introduction
Computing architectures are getting increasingly distributed, from multiple cores in one processor and multiple processors in a computing node, to many computing nodes. This creates a stronger demand for distributed and concurrent programming, because sequential programming models are not suitable for such architectures.

Distributed and concurrent programming is hard and largely different from sequential programming. Programmers have more concerns when it comes to taming parallelism. Distributed and concurrent programs are usually bigger than equivalent sequential ones. Models of distributed and concurrent programming languages are different from familiar and popular sequential languages [1,2].

Message passing models seem be the more appropriate solution because they the sharing of data with the exchange of messages. One of the well-known theoretical and practical models of message passing is the actor model. Using such a model, programs become collections of independent active objects (actors) that exchange messages and have no mutable shared state [3,4,5]. Actors can help developers to avoid issues such as deadlock, livelock and starvation, which are common problems for shared memory based approaches. There are a multitude of actor oriented libraries and languages, and each of them implements some variant of actor semantics. However, such libraries and languages either are designed on a simpler and easier model and using thread-based programming or use the event-based programming model, which is far more practical to develop large and efficient concurrent systems.

Simulation models are increasingly being used to solve problems and to aid in decision-making. The size and complexity of systems which are usually modeled (e.g., communication networks, biological systems, weather forecasting, manufacturing systems, etc.) are ever increasing. Modeling and simulation of such systems is challenging in that it requires suitable and efficient simulation tools that take advantage of the power of current computing architectures and, of course, of programming languages and software frameworks that can exploit such kinds of architecture and that offer the features useful for the development of such kinds of system.

In particular, agent-based modeling and simulation (ABMS) tools and techniques seem be the most suitable means to exploit the power of such computing architectures [6,7].

This paper presents an actor based software framework, called CoDE (Concurrent Development Environment), that has the features for simplifying the development of large and distributed complex systems and for guarantying an efficient execution of applications. In particular, CoDE has been experimented with success in the agent-based simulation. The next section introduces related
work. Section three describes the software framework. Section four describes how it can be used for agent-based simulation. Finally, section five concludes the paper by discussing its main features and the directions for future work.

2 Related Work
Several actor-oriented libraries and languages have been proposed in last decades and a large part of them uses Java as implementation language. A lot of work has also been done in the field of agent-based modeling and simulation. Moreover, some researchers used the actor model for the modeling and simulation of complex system. The rest of the section presents some of the most interesting works presented in the previous three fields.

Salsa [8] is an actor-based language for mobile and Internet computing that provides three significant mechanisms based on actor model: token-passing continuations, join continuations, and first-class continuations. In Salsa each actor has its own thread, and so scalability is limited. Moreover, message-passing performance suffers from the overhead of reflective method calls.

Kilim [9] is a framework used to create robust and massively concurrent actor systems in Java. It takes advantage of code annotations and of a byte-code post-processor to simplify the writing of the code. However, it provides only a very simplified implementation of an actor model where each actor (called task in Kilim) has a mailbox and a method defining its behavior. Moreover, it does not provide remote messaging capabilities.

Scala [10] is an object-oriented and functional programming language that provides an implementation of the actor model unifying thread based and event based programming models. In fact, in Scala an actor can suspend with a full thread stack (receive) or it can suspend with just a continuation closure (react). Therefore, scalability can be obtained by sacrificing program simplicity. In particular, Scala has been used for the development of Scalation [11]. Scalation is a domain specific language that supports multi-paradigm simulation modeling, including dynamics, activity, event, process and state oriented models. Akka [12] is an alternative toolkit and runtime system for developing event-based actors in Scala, but also providing APIs for developing in Java. One of its distinguishing features is the hierarchical organization of actors, so that a parent actor that creates some children actors is responsible for handling their failures.

Jetlang [13] provides a high performance Java threading library that should be used for message based concurrency. The library is designed specifically for high performance in-memory messaging and does not provide remote messaging capabilities.

Swarm [14] is the ancestor of many of the current ABMS platforms. The basic architecture of Swarm is the simulation of collections of concurrently interacting agents, and this paradigm is extended into the coding, including agent inspector actions as part of the set of agents. So in order to inspect one agent on the display, you must use another hidden, non-interacting agent. Swarm is a stable platform, and seems particularly suited to hierarchical models. Moreover, it supports good mechanisms for structure formation using multi-level feedback between agents, groups of agents, and the environment (all treated as agents).

Repast [15] is a well-established ABMS platform with many advanced features. It started as a Java implementation of the Swarm toolkit, but rapidly expanded to provide a very full featured toolkit for ABMS. Although full use of the toolkit requires Java programming skills, the facilities of the last implementations allow the development of simple models with little programming experience [16].

MASON [17] is a Java ABMS tool designed to be flexible enough to be used for a wide range of simulations, but with a special emphasis on “swarm” simulations of a very many (up to millions of) agents. MASON is based on a fast, orthogonal, software library to which an experienced Java programmer can easily add features for developing and simulating models in specific domains.

ATC [18] is a framework for the modeling and validation of real-time concurrent systems based on the actor model. In particular, it inherits all the functional capabilities of actors and further allows the expression of most of the temporal constraints pertaining to real-time systems: exceptions, delays and emergencies.

The Adaptive Actor Architecture [19] is an actor-based software infrastructure designed to support the construction of large-scale multi-agent applications by exploiting distributed computing techniques to efficiently distribute agents across a distributed network of computers. This software infrastructure uses several optimizing techniques to address three fundamental problems related to agent communication between nodes: agent distribution, service agent discovery and message passing for mobile agents.

An actor-based infrastructure for distributing RePast models [15] is proposed in [20].
solution allows, with minimal changes, to address very large and reconfigurable models whose computational needs (in space and time) can be difficult to satisfy on a single machine. Novel in the approach is an exploitation of a lean actor infrastructure implemented in Java. In particular, actors bring to RePast agents migration, location-transparent naming, efficient communication, and a control-centric framework.

Statechart actors [21] are an implementation of the actor computational model that can be used for building a multi-agent architecture suitable for the distributed simulation of discrete event systems whose entities have a complex dynamic behavior. Complexity is dealt with by specifying the behavior of actors through “distilled” statecharts [22]. Distribution is supported by the theatre architecture [23]. This architecture allows the decomposition of a large system into sub-systems (theatres) each hosting a collection of application actors, allocated for execution on to a physical processor.

3 Software Framework
CoDE (Concurrent Development Environment), is an actor based software framework that has the goal of both simplifying the development of large and distributed complex systems and guarantying an efficient execution of applications.

This software framework provides a concise actor model, which of course derives from the Hewitt and Agha’s actor model [3,5], and an implementation that supports the development of flexible and scalable distributed applications in which is possible to choose:
- The types of communication technologies supporting the exchange of messages among the actors running on different computation nodes.
- The threading solution used for executing the actors running on the same computation nodes.
- The additional services (e.g., messages broadcast and logging) that may be useful to simplify the developing of applications, to support their debugging and/or to increase their performances.

3.1 Software Model
A CoDE system is based on a set of interacting actors that perform tasks concurrently.

An actor is an autonomous concurrent object, which interacts with other actors by exchanging asynchronous messages. Communication between actors is buffered: incoming messages are stored in a mailbox until the actor is ready to process them.

Each actor has a unique mail address that is used to specify a target for communication. After its creation, an actor can change its behavior several times, until it terminates. Each behavior processes a set of specific messages. Therefore, if an unexpected message arrives, then it is maintained in the actor mailbox until a next behavior will be able to process it.

An actor can perform five types of action: (i) it can send messages to other actors or to itself, (ii) it can create new actors, (iii) it can update its local state, (iv) it can change its behavior, and (v) it can kill itself.

An actor has not direct access to the local state of the other actors and can exchange data with other actors only when it creates them or when it sends them a message. To limit the problems of the concurrent execution of the actors, the most known actor models impose the exchange of immutable messages. Of course, CoDE allows the exchange of immutable messages, but does not do force it because actors code must be written in Java and Java does not provide any direct support for immutability. However, it is possible to develop an implementation of the actor model that can guarantee (or check) if both the data, passed to another actor during the creation, and the messages, exchanged among the actors of an application, are immutable.

An actor can send messages only to the actors of which it knows the address, that is, the actors it created and the actors of which it received their addresses through a message. Moreover, an actor can send messages, can require an answer to them, and can label them as replies to the messages of other actors.

An actor has not explicit actions for the reception of messages, but its implementation will autonomously manage the reception of messages and then will execute the actions for their processing.

An actor has the possibility of setting and then modifying a timeout within its current behavior must receive a message. However, it has not explicit actions for managing the firing of such a timeout: even in this case, its implementation will autonomously observe the firing of the timeout and then will execute the actions for its management.

Depending on the complexity of the system and on the availability of computing and communication resources, actors can be aggregated in one or more actor spaces.
An actor space is a special actor that acts as a container of a set of actors. In particular, an actor space supports a transparent communication between local actors and remote actors (i.e., the actors of the other actor spaces) and enhances their functionalities through a set of services (e.g., the broadcasting of messages to local actors and the migration of local actors to other actor spaces).

Therefore, actors can require the execution of such services by sending a message to an actor space. It is possible because actors know a priori the address of their actor space and it provides the address of the other actor spaces of the application.

3.2 Software Implementation
CoDE is implemented by using the Java language and takes advantage of preexistent Java software libraries and solutions for supporting concurrency and distribution.

The architecture of a CoDE application can be divided in an application and a runtime layer. The application layer is represented by the behaviors and cases driving the execution of the actors of the application. The runtime layer implements the CoDE middleware infrastructures to support the development of applications based on several actors distributed on a set of different actor spaces.

The current implementation, besides building the components of the runtime layer, provides some components (i.e., interfaces, concrete and abstract classes) to simplify the development of the actors of an application.

The design of the software has been planned with the goal of guaranteeing an easy development of several implementations of the components of the runtime layer that take advantages of different technologies and algorithms without the need of modifications to the application layer (i.e., the code of the actors). This feature is important for enabling the use of different solutions for the management of the execution of the actors running inside the same actor space, but also for supporting the communication among actors of different actor spaces.

The most important components that contribute to flexibility of the CoDE software framework are the actor execution manager and the actor scheduler.

The actor execution manager defines the threading solution used for executing the actor, manages its life-cycle, passes the incoming messages to the actor behaviors and performs their actions. In particular, CoDE allows the definition of two types of actor execution manager: (i) active, i.e., each actor execution manager has its own thread of execution and (ii) passive, i.e., some actor execution managers share the same thread of execution.

The actor scheduler drives the concurrent execution of the actors of an actor space. Of course, their implementation mainly depends on the type of actor execution managers used in the application. In fact, when the actor execution managers of the application are active, then the scheduler delegates the large part of its duties to the Java threading and concurrency libraries. When the actor execution managers of the application are passive, then the scheduler completely manage their concurrent execution.

4 Agent-Based Simulation
The features of the actor model and the flexibility of its implementation make CoDE suitable for building agent-based modeling and simulation tools. In fact, the use of passive actors allows the development of applications involving large number of actors, and the possibility of using different schedulers for their execution allows the development of schedulers that are specialized for some specific application domains.

In particular, we started to cope with the simulation of some of the most known spatial models (e.g., the game of life [24], prey–predator [25] and boids [26]) by using both a continuous and a discrete representation of the space.

To support the simulation and the analysis of such models we need: (i) the synchronous execution of all the simulated individuals, and (ii) the possibility to have a graphical visualization of the simulation and the logging of the simulation data.

In such a model, the behavior of an individual is very simple: it gets information about the world (i.e., about the other individuals) and then performs some actions. Therefore, the simulation is also very simple if the individuals have direct access to the information about the world. It does not happen in an actor-based system where the individuals can share information by exchanging asynchronous messages. To make possible the simulation and easy the synchronization among the individuals, we develop a new actor execution manager and a new actors scheduler. The scheduler executes repeatedly all the individuals and after each execution step broadcasts them a “clock” message. The actor execution manager is a passive execution manager that divides the actor execution in steps in which the actor processes all the messages up to a “clock” message of the scheduler. Moreover, when it is processing the “clock” message then it can change its behavior.
While the logging of execution data is a basic feature that CoDE offers to its applications, we implemented two different GUIs for visualizing simulation based on continuous and discrete representations of the space. The main features of such GUIs is that they provide the visualization of the simulation through the logging data. Moreover, the logging data can be used for performing both qualitative and numeric analyses of the simulations. Fig. 1 shows a view of the GUI for discrete representations of the space during a simulation of the game of life model.

Fig. 1. GUI showing the simulation of the game of life.

<table>
<thead>
<tr>
<th>Actors number</th>
<th>100 cycles</th>
<th>1000 cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>6,953,207</td>
<td>9,221,430</td>
</tr>
<tr>
<td>500</td>
<td>17,365,659</td>
<td>42,728,500</td>
</tr>
<tr>
<td>1,000</td>
<td>26,855,399</td>
<td>101,550,196</td>
</tr>
<tr>
<td>5,000</td>
<td>148,404,992</td>
<td>616,903,010</td>
</tr>
<tr>
<td>10,000</td>
<td>474,400,876</td>
<td>1,695,905,585</td>
</tr>
<tr>
<td>50,000</td>
<td>7,676,558,254</td>
<td>16,804,178,726</td>
</tr>
<tr>
<td>100,000</td>
<td>27,847,382,628</td>
<td>44,747,352,517</td>
</tr>
<tr>
<td>500,000</td>
<td>532,079,637,390</td>
<td>699,710,824,444</td>
</tr>
<tr>
<td>1,000,000</td>
<td>2,123,332,712,806</td>
<td>2,606,157,257,685</td>
</tr>
</tbody>
</table>

Table 1. Execution times for two game of life model simulations.

This experimentation gave the opportunity to make some measures of the performance of the CoDE framework for agent-based simulations involving different numbers of actors. Table 1 presents the execution times of two simulations of the “game of life” having a length of respectively a hundred and a thousand of cycles and involving from few hundreds to more than a million of actors. These results were obtained on a laptop with an Intel Core 2 - 2.80GHz processor, 8 GB RAM, Windows 7 OS and Java 7 with 4 GB heap size. Moreover, all the performed experimentations showed that the use of the passive threading solution provides better performances than the active one even with a few number of actors.

5 Conclusion

This paper presented a software framework, called CoDE, which allows the development of efficient large systems by combining the sharing of threads among the actors of the systems with the delegation of the management of the reception of messages to the execution environment.

CoDE is implemented by using the Java language and is an evolution of ASIDE [27] with which it shares the same concise actor model. However, CoDE provides a simplified and more flexible implementation of the actor model that improves its usability for agent-based simulation. Moreover, it supports both file-based and code-based configuration of applications, and allows the logging of execution data in different formats and on different devices.

Future research activities will be dedicated, besides to continue the experimentation and validation of the software framework in the agent-based simulation, to the modeling and analysis of social networks and for the analysis of large and distributed data sets. Regarding the improvement and the extension of the software framework, current activities are dedicated to: (i) allow the mobility of actors between different actor spaces, (ii) provide a passive threading solution that fully take advantage of the features of multi-core processors, (iii) define a set of actor schedulers and supporting tools for performing simulations and analyzing the results of simulations in some specific application domains, (iv) improve the scalability of simulations on the number of actors through the definition of distributed actor schedulers, (v) provide a set of actor space services to simplify the coordination among the actors of a system (e.g., group communication protocols) and to enable the interoperability with Web services and legacy systems, and (vi) extend the adaptability of developed systems by managing the exchange of messages between actors through compositional filters [28].

References:


