A Model for Active Entities in Software Systems

Manojit Sarkar

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-95-09
April 1995

A Model for Active Entities in Software Systems

Manojit Sarkar
Computer Science Department
Brown University, Providence, RI 02912-1910
ms@cs.brown.edu

Abstract

We develop a model for active entities in software systems. An active entity, or simply an entity, is an active object with a private state and one or more independent threads of control. Entities communicate with each other by message passing. A collection of communicating entities constitute a concurrent and distributed system. We present a language for describing entities. The semantics and implementation of the language are also discussed.

1 Introduction

Concurrency and distribution in software systems are already widespread today and are becoming more widespread each day. A distributed system typically consists of a number of active autonomous entities communicating with each other by message passing. In this paper, we discuss what constitutes an entity and present a language for describing them. An active entity, or simply an entity, is modeled as an object with a private state and one or more independent threads of control.

2 Related Research

A Unix *process* [3] is a good example of an entity. Unix based distributed systems are usually built using multiple processes. Traditionally Unix processes were single-threaded, but modern Unix processes can be multi-threaded. None of the programming languages, however, recognize processes. Our language recognizes multi-threaded entities and provides a higher level model for entities than the process model of Unix.

The *actor* model was first developed by Hewitt [5] to model active entities. It was further developed by Agha [1, 2]. Our entity model have some similarities with the actor model. Agha modeled actors as autonomous and concurrently executing objects which send each other messages asynchronously. In response to receiving a message, an actor can send another message asynchronously, change its state before processing the next message, or create a new actor. Several other concurrent object-oriented programming languages have been proposed recently [4, 8, 9, 10].

Liskov and Scheifler [6, 7] developed a system and a programming language called *Argus* for building distributed systems under some special problems such as network partitions and crashes of remote nodes. In a distributed program constructed with Argus, the *guardian* and the *handler call* provide functionalities that are similar to the functionalities provided by entities and message passing in our model respectively. Entities, unlike guardians however, are not designed to guard against special network failures. Argus also provide support for atomic activities through *actions* which are serializable and which either commits or aborts completely. The model described in this paper do not attempt to provide support for atomic transactions.

3 Model for Entities

Entities encapsulate state, behavior and threads of control. They communicate with each other by message passing. Each entity has an unbounded *queue* for receiving messages and an *listener thread* for processing *messages*. We will commonly refer to the listener thread as the *listener*. The listener may create other threads of control for carrying out actions concurrently. These threads are called *action threads*. All threads within a single entity share a single state and can modify it.

The model also includes conventional *objects* to represent passive entities. Objects, unlike entities, do not have their own threads of control. Objects are like objects of conventional object-oriented languages such as *C++* and *Modula-3*.

4 Messages

A *message* is a sequence of named variables, called *fields* of the message. Different fields can have different types. The expression *m.f* designates the field named *f* in the message *m*. Each message has a predefined field called `depends_on` which can hold a pointer to a function with signature `Boolean func_name(Message)`. The use of this field will be discussed shortly. Each message also has a *type*. Messages are divided in two categories: *commands* and *requests.*

4.1 Commands

Commands are used for asynchronous communication. The sender of a command does not wait for a response; it continues to execute after sending the command. The receiver may perform some operation or ignore the command. The following statement sends the command `c` to the entity `e`:

```
send c to e
```

The declaration of a command type has the form:

```
type T1 = command (f1; f2;...; fn)
```

where T_1 is the type name and $f_1, f_2, ..., f_n$ are field declarations of the form:

```
<field_type> <field_name>
```

Here is an example of a command type declaration:

```
type Press = command (Btn btn; Float
    x; Float y;)
```

4.2 Requests

Requests are used for *synchronous* communication. The receiver must send a reply in response to a request. Sending a request blocks the sender which resumes when the reply is received. The receiver may overwrite certain designated fields of the request message and send it back to the original sender by executing a *reply* statement. The following statement sends the request `r` to the entity `e` and causes the sender to block:

```
request r to e
```

The reply statement executed by the receiving entity looks like this:

```
reply r
```

The receiver may also send an additional result along with the modified request message by executing the following:

```
reply r with result s
```

Request and its reply with optional result provide the remote procedure call abstraction. The run-time system associates a sender with a request when the request is sent. This information is used later to reply to the original sender. The receiver may send the request to another entity before sending it back and so on.

A request type is declared like this:

```
type T1 = request (f1; f2;...; fn)
    [with result type T2]
```

where T_2 is any type and $f_1, f_2, ..., f_n$ are the field declarations. The type T_2 has been enclosed in square braces to indicate that it is optional. This convention is followed henceforth to indicate optional items. A request field declaration has the form:

```
<qualifier> <field_type> <field_name>
```

There are three qualifiers: `in`, `out` and `inout`. The receiver can return information by overwriting the request fields that are qualified as either `out` or `inout`.

A typical request-reply pair in a user interface system may be the following:

```
type Pick = request (in Float x;
    in Float y; in Integer max_depth;
    in Integer depth;out Pick p)
    with result type Boolean
```

4.3 Message Construction

A message constructor expression has the form:

```
<msg_type> (<bindings>)
```

where `<bindings>` is a list of keyword binding of the form:

```
<field_name> := <expr>
```

Here is an example of a command constructor:

```
Press(btn := 1st, x := 2.0, y := 3.0)
```

4.4 Dependence Among Messages

The listener thread of the entity processes incoming messages. When the listener removes a message from the queue, the message is said to have been *scheduled*. A message m_1 is said to depend on another message m_2, if the receiver must *schedule* m_2 before scheduling m_1. The sender can specify dependence relationships between messages by initializing the `depends_on` field of a message m_1 to point to an appropriate function. The function must take a message m_2 as argument and return a boolean value which indicates if m_1 depends on m_2 or not. The listener evaluates the function with each message that precedes m_1 in the message queue.

This scheme allows the programmer to specify dependence relationships at run-time. Therefore it is possible to control both presence and absence of such relationships based on the dynamic context. As an example consider the following two message types: `AddToBalance` and `GetBalance`. If an up-to-date balance is needed, the programmer may defer scheduling the `GetBalance` message till all the preceding `AddToBalance` messages are processed. This can be done by sending a request that depends on any message of type `AddToBalance`.

On the other hand, if an approximate balance suffices, the programmer may want to schedule both messages as soon as possible. The language makes provisions for both situations.

The scheme above allows a message m_1 to depend on another message m_2 if and only if m_2 is received before m_1. A choice can also be made as to whether a message can be allowed to depend on a message that arrives at a later time; perhaps bounded by a maximum waiting time. It is not clear yet if we need so much expressiveness and if these are the best ways to achieve that expressiveness.

5 Entities

An entity has a state, a behavior and one or more independent execution threads.

5.1 State

The state of an entity consists of a control part and a data part. The control state and the data state of an entity consist of a sequence of *control fields* and a sequence of *data fields respectively*. The purpose of separating control fields from data fields will become clear shortly. The fields are declared as follows:

```
type T = entity (
  control fields: c1; c2;...; cn;
  data fields: d1; d2;...; dn;)
```

where $c_1, c_2, \ldots, c_n, d_1, d_2, \ldots, d_n$ are field declarations.

5.2 Behavior

The behavior of an entity is specified by a sequence of *triggers*. An entity type with both state and behavior declarations looks like this:

```
type T1 = entity (control fields:...;
  triggers: t1; t2; ...; tn;
  data fields:...;)
```

where $t_1, t_2, \ldots, t_n$ are the trigger declarations. A trigger declaration has the form:

```
<msg_type> <msg_name> <cond>
  <prologue> <body>
```

where <cond> is a boolean expression and <prologue> and <body> are code segments. The prologue and the body together constitute the *action* part of the trigger. The clause <msg_type> is a message type name. A message of only this type may *fire* the trigger. The clause <msg_name> is an identifier for accessing the message and its fields in code segments of the action.

5.3 Action Priority

The priority is a value of a subrange type. The subrange type and the priority type of an entity type is declared like this:

```
type T1 = subrange (<low>...<high>)

type T2 = entity (mode type...
  priority type = T1;
  control fields:...;
  triggers:...;
  data fields:...)
```

where <low> and <high> are two ordinal values with the same *base* type, called the base type of the subrange. An ordinal valued expression <priority> of the priority type is included in a trigger as follows:

```
<msg_type> <msg_name> <cond>
  <prologue> <priority> <body>
```

where <priority> must evaluate to an ordinal value within the specified subrange. The expression can refer to any control field but not data fields. The value returned by <priority> determines the priority of the action thread that executes the body.

6 Message Processing Algorithm

Sending a message to an entity appends the message to the queue. The listener reads messages from the queue and processes them by firing appropriate triggers. The condition acts as a guard. A trigger cannot be fired unless the guard is satisfied. Also, a message can be used to fire a trigger only if the message does not depend on any preceding message still pending in the queue. The listener verifies this by evaluating the depends_on function against all pending messages preceding the message.

Once a trigger is fired, the prologue is executed by the listener and then the body is invoked in a separate action thread by the listener. Hence the listener is free to process other messages after executing the prologue. The listener and the action threads execute concurrently. The control fields can be accessed by the listener thread while evaluating the condition and executing the prologue. The control fields can also be accessed by the action threads while executing the body. The data fields are accessed by the action threads only.

Execution of the prologue is assumed to take a small and bounded amount of time. Also, the action threads may lock control fields only for small and bounded amounts of time. These restrictions allow the listener thread to process messages with bounded delay. This guarantees that every message will be *serviced* within at most *nk* time units where *n* is the number of pending messages in the queue and *k* is the maximum time required to service any message.

The listener thread processes messages by executing the following loop for the entire life of the entity:

```
while (alive) do
   acquire(lock);
   while (!new_msg_arrived &
   !control_fields_changed)
      wait(lock,cond);
   endwhile;
   service_pending_messages();
   new_msg_arrived = false;
   control_fields_changed = false;
   release(lock);
endwhile
```

The procedure `service_pending_messages()` is as follows:

```
for (m := first_pending_msg to
   last_pending_msg)
   if (m's type matches no trigger's
   message type)
      discard m
   elsif (m can fire a trigger)
      if (m doesn't depend on any msg)
         pick the 1st fireable trigger
         execute prologue
         invoke body in new thread
      elsif (m depends on another msg)
         keep m pending
      endif
   elsif (m cannot fire any trigger)
      if (m doesn't depend on any msg)
         discard m
      elsif (m depends on another msg)
         keep m pending
      endif
   endif
endfor
```

The algorithm keeps message *m* pending when *m* depends on a preceding pending message even though *m* failed to fire any trigger. The reason is as follows. It is possible that the programmer intended to delay *m* until some of *m*'s preceding messages are processed. Only way to achieve this is to make *m* fail all triggers. However if the programmer intended to discard *m*, then he could add a new trigger (with lowest preference) which would fire but have null action associated with it. This would have the same effect as discarding *m* when *m* fails to fire all the other triggers.

Each message is either discarded or it fires exactly one trigger. The same trigger can be fired by more than one message and each firing creates a separate *activation instance* of the trigger. Once a message fires a trigger it is considered scheduled and is removed from the queue. After the execution of the trigger's action the message is said to be *consumed*.

When a message arrives, the run-time system gives it an *arrival number*, which is a an integer greater than zero. The message's *arrival time* and *sender are* also registered with the message by the run-time system. The arrival number, the arrival time and the sender information are used by the run-time system later. The arrival time, the arrival number and the sender of a newly constructed message are *0.0*, *0* and `nil` respectively.

The following is executed by the sender while sending a message:

```
acquire(lock);
   enqueue the message
   arrival_no++;
   new_msg_arrived := true;
   signal(cond);
   release(lock);
return arrival_no;
```

The following is executed by the action body whenever it updates any control field:

```
acquire(lock);
   update control field
   control_fields_changed := true;
   signal(cond);
release(lock);
```

7 Remaining Issues

Our model does not provide support for atomic transactions. We wish to incorporate atomic transactions in our model. Multicasting is a frequently required function in distributed systems and we would like to explore the best ways to incorporate it. We also want to investigate how to express and maintain constraints within concurrent and distributed systems. At a minimum, the model should be able support one way dependencies in a simple and efficient manner. Also, the implications of side-effects of trigger conditions and priority expressions have not been thought out fully. For the time being they are not guaranteed to be side-effect free.

8 Conclusions

Active entities play a central role in distributed systems and intra-entity concurrency is a very desirable property. In the Unix world, processes are threads are used to build concurrent active entities. Conventional programming languages however do not recognize these abstractions. This paper has taken a step towards filling this vacuum with a model and a specification language for active entities.

References

[1] Gul Agha. Concurrent object-oriented programming. *Comm. of the ACM,* 33(9), September, 1990.

[2] Gul Agha. *Actors: A model of Concurrent Computation is Distributed Systems*. The MIT Press. 1986.

[3] Maurice J. Bach. *The design of the Unix operating systems.* Prentice-Hall, Englewood Cliffs, NJ, USA, 1986.

[4] J. van den Bos and C. Laffra. PROCOL: A concurrent object-oriented language with protocols delegation and constraints. *Acta Informatica* 28:511-538. 1991.

[5] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.

[6] Barbara Liskov. Distributed programming in Argus. *Comm. of the ACM, 31(3)*. March, 1988.

[7] Barbara Liskov and Robert Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Trans. on Programming Languages and Systems*, 5(3):381–404, July, 1993

[8] Chris Tomlinson and Vineet Singh. Inheritance and synchronization with enabled-sets. OOPSLA 1989.

[9] Anand Tripathi, Eric Berge and Mehmet Aksit. An implementation of the object-oriented programming language SINA. *Software-Practice and Experience*, 19(3), March, 1989.

[10] Akinori Yonezawa. *ABCL: An Object-Oriented Comcurrent System.* Computer Systems Series. The MIT Press. 1990.