Implementing Real-Time Actors with MetaJava

Michael Golm, Jürgen Kleinöder

April 1997 TR-I4-97-09

Technical Report

Computer Science
Department

Operating Systems — IMMD IV

Friedrich-Alexander-University
Erlangen-Nürnberg, Germany

Presented at the ECOOP 97 Workshop on Reflective Real-time Object-Oriented Programming and Systems, June 10, 1997, Jyväskylä, Finland

Implementing Real-Time Actors with MetaJava

Michael Golm, Jürgen Kleinöder[1]
University of Erlangen-Nürnberg, Dept. of Computer Science IV
Martensstr. 1, D-91058 Erlangen, Germany
{golm, kleinoeder}@informatik.uni-erlangen.de

Abstract

Actors are a suitable abstraction to manage concurrency in real-time applications. Meta-level programming can help to separate real-time concerns from application concerns. We use reflection to transform passive objects into active objects. Then we extend the meta-level implementation of the actors to be sensitive to soft real-time requirements.

1 Introduction

Meta-level interfaces allow the service provided by a base-level API to be adjusted to specific application needs and run-time environments. MetaJava [KlG96a] extends the Java Virtual Machine by a meta-level interface (MLI). The MetaJava MLI allows metaobjects to modify the interpreter's object model. We show, how the meta-level interface can be used to implement active objects. Active objects, or actors, are an appropriate abstraction to manage concurrency in real-time systems. Most real-time systems have a reactive nature. They respond to signals from sensors and do control actuators. The signal/response behavior of real-time systems maps well with the message/reply scheme of actors. To be useful in an environment with real-time constraints the original developed actor system must be extended. The proposed actor implementation is not intended for hard real-time systems. To satisfy hard real-time constraints, it is necessary to find out worst-case execution times, use incremental garbage collection, use resource negotiation, etc. This was investigated in RT-Java[Nil96].

The paper is structured as follows. Section 2 introduces relevant concepts of MetaJava. Section 3 discusses the actor model. Section 4 explains the actor implementation and Section 5 the real-time extensions to this implementation. Section 7 discusses related work and Section 8 concludes the paper.

2 MetaJava

MetaJava is an extension to the Java Virtual Machine [LiY96] that supports structural and behavioral reflection [Fer89] in Java. The base-level object model is the Java model. MetaJava provides mechanisms to modify this object model and to add extensions - for example, persistent objects, remote objects, replicated objects, or active objects.

Base-level objects and meta-level objects are defined separately. Meta-level objects which inherit from the class `MetaObject` can be attached to base-level objects. After a metaobject is attached, it can register for events of base-level computation (lines 3,4 of Figure 1). Operations that could raise an event include method invocation, variable access, object creation, and class loading. An event description contains sufficient information about the event and enables the metaobject to reimplement the event-generating operation. A method-event description, for example, contains the following information:

- a reference to the called object
- the method name and method signature
- the method arguments

An event is delivered to the event-handler method of the attached metaobject (line 5 of Figure 1). This method is responsible for an appropriate implementation of the operation. It could continue with the default mechanism or define a customization. The default mechanism for method executions is provided with the method `continueExecutionVoid` (line 6 of Figure 1).

When attaching a metaobject to a base-level object, base level and meta level are visible, but afterwards the meta level is transparent. During this process information about the semantics of the base-level object can be passed to the metaobject. This information consists of information about methods, instance variables, and other object properties. The names of those methods or instance variables must be

1. This work is supported by the *Deutsche Forschungsgemeinschaft DFG* Grant *Sonderforschungsbereich SFB 182*, Project *B2*.

```
public class MetaObject {
    protected void attachObject(Object baseobject) { ... }
    protected void registerEventMethodCall(Object baseobject) { ... }
    protected void registerEventMethodCall(Object baseobject, String methods[]) { ... }
    public void eventMethodEnterVoid(Object o, EventDescMethodCall event) { ... }
    protected void continueExecutionVoid(Object baseobject, EventDescMethodCall event) { ... }
    ...
}
```

Figure 1: The MetaObject class

passed to the metaobject as strings[2]. Once the metaobject has been attached, the meta level is transparent to the base-level object.

3 Actors

The actor model, developed by Hewitt [Hew77] and Agha [Agh86], is a approach to manage concurrency. Recently, it has been applied to the domain of real-time programming [SGW96], [RAS96].

One advantage of the actor model is the easy synchronization. In the original actor model, there is exactly one thread active in one actor and thus there is no need to synchronize inside an actor. However, in multiprocessor real-time systems it can be beneficial to have more than one thread execute in an actor - for example, if the threads execute read-only methods.

Our actor model differs from the original actor model in two points: message passing is not asynchronous and there can be an arbitrary number of threads executing in an actor.

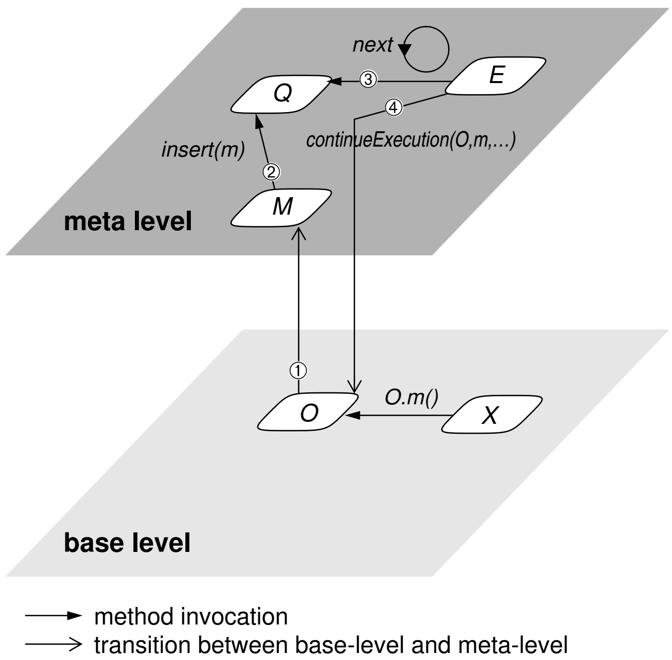
The use of actors leads to a very dynamic and adaptable system. Actors are a means to implement a *best-effort* real-time system - that is, a system that tries to meet timing constraints but can not guarantee this a priori. Actor systems are not intended for hard real-time systems with guaranteed timing behavior.

4 Implementing actors at the meta level

Active objects are an extension of the passive object model. A passive object implements the functional aspect of the actor. The actor behavior is implemented at the meta level, as shown in Figure 2. The MetaActive metaobject transforms a passive object into an actor. The constructor of MetaActive configures its state according to the parameters and attaches itself to the base-level object.

2. The current version of MetaJava uses a standard Java compiler. This means that there is no linguistic support for reflective programming.

Figure 3 shows a part of the implementation of the MetaActive metaobject. The constructor initializes the active object execution environment, consisting of QueueManager and Executor (lines 5 to 8 of Figure 3) and attaches itself to the base-level object. MetaActive reimplements the method call mechanism to support the actor behavior.



Object *X* calls method *m* of the active object *O*. This base-level action is implemented at the meta level as follows: ① Meta object *MetaActive* is attached to *O* and thus it receives the *method-enter* event. ② *MetaActive* inserts the message in *Queue*.

Until the message is inserted into the queue, all operations are performed by the caller thread. Then the caller thread is blocked until the *Executer* returns from the *continueExecution* call.③ The *Executer* receives a message from *Queue*. ④ It invokes the appropriate method at the base-level object *O* using *continueExecution*.

Figure 2: Separation of base level and meta level in the actor implementation

```
1   public class MetaActive extends MetaObject
2   {
3       QueueManager queue_;
4       public MetaActive(Object obj) {
5           // init active object environment
6           queue_ = new FIFOQueueManager();
7           Executor executor = new Executor(queue_,obj);
8           (new Thread(executor)).start();
9           // establish base-meta link
10          attachObject(obj);
11          registerEventMethodCall(obj);
12      }
13      public void eventMethodEnterVoid(Object o, EventDescMethodCall event) {
14          ActorMessage msg = new ActorMessage(event);
15          queue_.insert(msg);
16          synchronized(msg) { try { msg.wait(); } catch(InterruptedException e) {...} }
17      }
18      public Object eventMethodEnterObject(Object o, EventDescMethodCall event) {
19          ActorMessage msg = new ActorMessage(event);
20          queue_.insert(msg);
21          synchronized(msg) { try { msg.wait(); } catch(InterruptedException e) {...} }
22          return msg.returnValueObject();
23      }
24      ...
25  }
```

Figure 3: The MetaActive class

When receiving a method-enter event MetaActive creates a new message object (line 14 of Figure 3) and inserts it into the message queue. Then the caller thread blocks (lines 16 and 21 of Figure 3) until it is notified by the Executor. The Executer object (Figure 4) continuously obtains messages

```
class Executor extends MetaObject implements Runnable
{
    QueueManager queue_;
    Object obj_;

    public Executor(QueueManager queue, Object obj) {
        queue_ = queue;
        obj_ = obj;
    }

    public void run() {
        for(;;) {
            ActorMessage msg = (ActorMessage) queue_.next()
            EventDescMethodCall event = msg.getEvent();
            continueExecutionVoid(obj_, event);
            msg.notifyAll();
    }
}
```

Figure 4: The Executer class

from the QueueManager. To enable different message scheduling policies, MetaActive and Executer only use the interface QueueManager (Figure 5). The FIFOQueueManager implements this interface and provides a first-in-first-out message queue.

```
public interface QueueManager {
    public void insert(Object event);
    public Object next();
}
```

Figure 5: The QueueManager interface

5 Real-time extensions to the actor metaobject

To handle real-time requirements, the actor metaobject developed in the previous section must be extended to include temporal considerations. We consider the following real-time related aspects of actors:

(1) The policy to accept messages and insert them into the message queue.

This includes the policy to assign priorities to messages. Priorities can be based on the message name, the message sender or the message receiver.

(2) The policy to map messages to methods and execute them.

Some methods may need wrapper functions that check pre- and post-conditions. In special situation it may be possible to use a separate thread to execute the method.

(3) The synchronization policy.

If multiple threads are active in one actor, a synchronization policy for these actors is needed.

(4) The policy to control method execution.

It is possible to specify a maximum time quantum for method execution. If this time is exceeded, the method execution is aborted. Aborting a method could lead to an inconsistent object state. Therefore, after aborting a method, another method to clean-up must be invoked.

This considerations are implemented by a different actor metaobject. This metaobject is parametrized with information about message properties. A message property consists of

- the message name
- the name of methods which must be invoked before and after the called method
- the message priority
- the maximum time quantum of the message, including wrapper functions

As shown in Figure 6, only the metaobject constructor must be reimplemented to initialize a new execution environment for active objects with real-time properties.

The priority in the message-property specification is used by the insert method of RTQueueManager for a placement decision. The RTExecutor (Figure 7) uses the message property to execute the wrapper functions (lines 19 and 21 of Figure 7) and control maximal execution times. To control maximal execution times, a watch-dog thread is started, which blocks until the time quantum is over and then sends the thread a stop signal. This causes the thread to throw a ThreadDeath exception which is catched (lines 22 to 25 of Figure 7) and so triggers the clean-up function.

```
public class MetaRTActive extends MetaActive
{
    public MetaRTActive(Object obj, MessageProperties props) {
        // init real-time active object execution environment
        queue_ = new RTQueueManager(props);
        RTExecutor executor = new RTExecutor(queue_,obj, props);
        (new Thread(executor)).start();
        // establish base-meta link
        attachObject(obj);
        registerEventMethodCall(obj);
    }
}
```

Figure 6: The MetaRTActive class

```
class RTExecutor extends Executor
{
    MessageProperties props_;

    public RTExecutor(QueueManager queue, Object o, MessageProperties props) {
        super(queue,obj);
        props_=props;
    }

    public void run() {
        EventDescMethodCall event;
        for(;;) {
            ActorMessage msg = (ActorMessage) queue_.next()
            EventDescMethodCall event = msg.getEvent();
            MessageProperty prop = props_.getProperty(event.methodname, event.signature);
            ... prepare the wrapper event descriptions ...
            ... initialize and start the watchdog thread ...
            try {
                doExecute(obj_, pre_wrapper_event);
                continueExecutionVoid(obj_, event);
                doExecute(obj_, post_wrapper_event);
                ... terminate watchdog thread ...
            } catch(ThreadDeath e) {
                ... prepare the cleanup_event description ...
                doExecute(obj_, cleanup_event);
            }
            msg.notifyAll();
        }
    }
}
```

Figure 7: The RTExecutor class

6 Related Work

The main difference between our work and other implementations of real-time actors is, that we do not need any support for actors from the run-time system. We solely rely on a minimal object model with reflective capabilities.

The actor model described in this paper was inspired by the Real-Time Object-Oriented Modeling language ROOM [SGW96]. ROOM's primary focus lies on the design of actor-based real-time systems. The actor model is predefined in the ROOM virtual machine layer and can not be extended by applications.

DROL [ToT92] is an actor implementation based on the ARTS kernel. It relies on kernel support for active objects, but can control the execution of actors with metaobjects. MetaJava provides means to implement an object model that supports active objects. This way, different active object semantics can co-exist within the same program.

RTsynchronizers [RAS96] extend an actor model with real-time constraints. These constraints on message executions are defined separately from the actor definition and thus the actor can be reused in a different environment. RTsynchronizers define conditions over time variables. These conditions must be fulfilled before a message is scheduled for execution. While the declarative approach of RTsynchronizers seems to improve reusability of timing constraints, it is not clear, how the declarations are compiled into an efficient executable form.

7 Conclusion and Future Work

We described a scheme to implement real-time sensitive actors using the reflective Java interpreter MetaJava. More work needs to be done of supporting well known policies for real-time sensitive actors. This includes message schedule times, periodic tasks, etc. It would be interesting to investigate the possibilities of combining the actor meta system with other metaobjects, such as MetaRemote [KlG96a] or MetaReplication [KlG96b].

8 References

Agh86 G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

Fer89 J. Ferber. Computational Reflection in class based Object-Oriented Languages. *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '89,* New Orleans, La., Oct. 1989, pp. 317–326.

Hew77 C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3) 1977, pp. 323-364

KlG96a J. Kleinöder, M. Golm. MetaJava: An Efficient Run-Time Meta Architecture for Java. *Proc. of the International Workshop on Object Orientation in Operating Systems - IWOOOS '96*, October 27-18, 1996, Seattle, Washington, IEEE, 1996.

KlG96b Jürgen Kleinöder, Michael Golm. *Transparent and Adaptable Object Replication Using a Reflective Java*, Tech. Report TR-I4-96-07, Universität Erlangen-Nürnberg: IMMD IV, Sept. 1996

LiY96 T. Lindholm, F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Sept. 1996.

MY93 S. Matsuoka, A. Yonezawa. Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages. in G. Agha, A. Yonezawa, P. Wegner. *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, 1993. pp. 107-150

Nil96 K. Nilsen. Issues in the Design and Implementation of Real-Time Java. in *Java Developer's Journal*, June 1996.

RAS96 Ren, G. Agha, Saito. A Modular Approach for Programming Distributed Real-Time Systems. Special Issue of the *Journal of Parallel and Distributed Computing* on Object-Oriented Real-Time Systems, 1996.

SGW96 B. Selic, G. Gullekson, P.T.Ward. *Real-Time Object-Oriented Modelling*. John Wiley & Sons, Inc., 1996.

ToT92 K. Takashio, M. Tokoro. DROL: An Object-Oriented Programming Language for Distributed Real-Time Systems. *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '92,* pp. 276-294