Programming and Deployment of Active Objects with Application-Level Scheduling[*]

ABSTRACT

We extend and implement a modeling language based on concurrent active objects with application-level scheduling policies. The language allows a programmer to assign priorities at the application level, for example, to method definitions and method invocations, and assign corresponding policies to the individual active objects for scheduling the messages. Thus, we leverage scheduling and performance related issues, which are becoming increasingly important in multi-core and cloud applications, from the underlying operating system to the application level. We describe a tool-set to transform models of active objects extended with application-level scheduling policies into Java. This tool-set allows a direct use of Java class libraries; thus, we obtain a full-fledged programming language based on active objects which allows for high-level control of deployment related issues.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.3.0 [**Programming Languages**]: General

General Terms

Languages, Design

Keywords

Application-level scheduling, Priority scheduling, Creol, Java, Actor model, Concurrent active objects

1. INTRODUCTION

One of the major challenges in the design of programming languages is to provide high-level support for multi-core and cloud applications which are becoming increasingly important. Both multi-core and cloud applications require an explicit and precise treatment of non-functional properties, e.g., resource requirements. On the cloud, services execute in the context of virtual resources, and the amount of resources actually available to a service is subject to change. Multi-core applications require techniques to help the programmer optimally use potentially many cores. At the operating system level, resource management is greatly affected by scheduling which is largely beyond the control of most existing high-level programming languages. Therefore, for optimal use of the available resources, we cannot avoid leveraging scheduling and performance related issues from the underlying operating system to the application level. However, the very nature of high-level languages is to provide suitable abstractions that hide implementation details from the programmer. The main challenge in designing programming languages for multi-core and cloud applications is to find a balance between these two conflicting requirements.

We investigate in this paper how concurrent active objects in a high-level object-oriented language can be used for high-level scheduling of resources . We use the notion of concurrent objects in Creol [9, 3]. A concurrent object in Creol has control over one processor; i.e. it has a single thread of execution that is controlled by the object itself. Creol processes never leave the enclosing object; method invocations result in a new process inside the target object. Thus, a concurrent object provides a natural basis for a deployment scheme where each object virtually possesses one processor. Creol further provides high-level mechanisms for synchronization of the method invocations in an object; however, the scheduling of the method invocations are left unspecified. Therefore, for the deployment of concurrent objects in Creol, we must, in the very first place, resolve the basic scheduling issue; i.e. which *method* in which *object* to select for execution. We show how to introduce priority-based scheduling of the messages of the individual objects at the application-level itself.

In this paper we also propose a tool architecture to deploy Creol applications. To prototype the tool architecture, we choose Java as it provides low-level concurrency features, i.e., threads, futures, etc., required for multi-core deployment of object-oriented applications. The tool architecture prototype transforms Creol's constructs for concurrency to their equivalent Java constructs available in the `java.util.-concurrent` package. As such, Creol provides a high-level structured programming discipline based on active objects on top of Java. Every active object in Creol is transformed

[*]This work is partially supported by the EU FP7-231620 project: HATS.

to an object in Java that uses a priority manager and scheduler to respond to the incoming messages from other objects. Besides, through this transformation, we allow the programmer to seamlessly use, in the original Creol program, Java's standard library including all the data types. Thus, our approach converts Creol from a modeling language to a full-fledged "programming" language.

Section 2 first provides an overview of the Creol language with application-level scheduling. In Section 3, we elaborate on the design of the tool-set and the prototype. The use of the tool-set is exemplified by a case study in Section 4. Section 5 summarizes the related work. Finally, we conclude in Section 6.

2. APPLICATION-LEVEL SCHEDULING

Creol [9] is a full-fledged object-oriented language with formal semantics for modeling and analysis of systems of concurrent objects. Some Creol features include interface and class inheritance and being strongly typed such that safety of dynamic class upgrades can be statically ensured [15]. In this section, we explain the concurrency model of Creol using a toy example: an exclusive resource, i.e., a resource that can be exclusively allocated to one object at a time, behaving as a mutual exclusion token. Further, we extend Creol with priority-based application-level scheduling.

The state of an object in Creol is initialized using the `init` method. Each object then starts its active behavior by executing its `run` method if defined. When receiving a method call, a new process is created to execute the method. Creol promotes *cooperative* non-preemptive scheduling for each active object. It means that a method runs to completion unless it explicitly releases the processor. As a result, there is no race condition between different processes accessing object variables. Release points can be conditional, e.g., **await** ~taken. If the guard at a release point evaluates to true, the process keeps the control, otherwise, it releases the processor and becomes disabled as long as the guard is not true. Whenever the processor is free, an enabled process is *nondeterministically* selected for execution, i.e., scheduling is left unspecified in standard Creol in favor of more abstract modeling.

To explain extending Creol with priority specification and scheduling, we take a *client/server* perspective. Each caller object is viewed as a client for the callee object who behaves as a server. We define priorities at the level of language constructs like method invocation or definition rather than low-level concepts like processes.

On the server side, an interface may introduce a *priority range* that is available to all clients. For instance, in Line 2 of `Resource` in Listing 1, we can define a priority range:

```
priority range 0..9
```

On the client side, method calls may be given priorities within the range specified in the server interface. For example, calling the `request` method of the `mutex` object:

```
mutex ! request() priority(7);
```

Scheduling *only* on the basis of client-side priority requirements is too restrictive. For example, if there are many `request` messages with high priorities and a low priority `release`, the server might block as it would fail to schedule the `release`. In this particular example, we can solve this problem by allowing the servers to prioritize their methods. This involves a declaration of a priority range generally

Listing 1: Exclusive Resource in Creol

```
 1 interface Resource begin
 2
 3    op request()
 4    op release()
 5 end
 6
 7 class ExclusiveResource implements Resource begin
 8   var taken := false;
 9
10   op request () ==
11     await ~taken;
12     taken := true;
13   op release () ==
14     taken := false
15 end
```

depending on the structure of the class. In our example, assuming a range 0..2 added in Line 9, this requires changing the method signatures in the `ExclusiveResource` class:

```
op request() priority(2) == ...
op release() priority(0) == ...
```

This gives `release` a higher priority over `request`, because by default, the smaller values indicate higher priorities. Furthermore, the server may also introduce priorities on certain characteristics of a method invocation such as the kind of "release statement" being executed. For example, a process waiting at Line 11 could be given a higher priority over new requests by assigning a smaller value:

```
await ~taken priority(1);
```

The priority can be specified in general as an expression; we used here only constant values. Evaluation of this expression at runtime should be within the given range. If no priority is specified for a method invocation or definition, a default priority value will be assigned.

We discussed different levels of application-level priorities. Note that now each method invocation in the queue of the object involves a tuple of priorities. We define a general function δ as an "abstract priority manager":

$$\delta : P_1 \times P_2 \times P_3 \times \ldots \times P_n \longrightarrow P$$

The function δ maps the different levels of priority in the object ($\{P_1, \ldots, P_n\}$) to a single priority value in P that is internally used by the object to order all the messages that are queued for execution. Each method in the object may have a δ assigned to it. In an extended version of δ, it can also involve the internal state of the object, e.g., the fields of the object. In this case, we have dynamic priorities.

For example, in `ExclusiveResource`, we have two different levels of priorities, namely the client-side and server-side priorities, which range over $P_1 = \{0, \ldots, 9\}$ and $P_2 = \{0, 1, 2\}$, respectively. So, we define $\delta : P_1 \times P_2 \to P$ as:

$$\delta(p_1, p_2) = p_1 + p_2 \times |P_1|$$

To see how it works, consider a `release` and a `request` message, both sent with the client side priority of 5. Considering the above method priorities, we have $\delta(5, \mathit{request}) = \delta(5, 2) = 25$ and $\delta(5, \mathit{release}) = \delta(5, 0) = 5$. It is obvious that the range of the final priority value is $P = \{0, \ldots, 29\}$.

Note that the abstract priority manager in general does *not* completely fix the "choice" of which method invocation to execute. In our tool-set, we include an extensible library

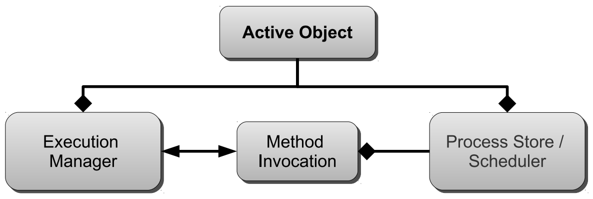

Figure 1: 𝒞risp Architecture: Structural Overview

of predefined scheduling policies such as strong or weak fairness that further refine the application-specific multi-level priority scheduling. The policies provided by the library are available to the user to annotate the classes. We may declare a scheduling policy for the `ExclusiveResource` class by adding at Line 9 in Listing 1:

```
scheduling policy StronglyFairScheduler;
```

A scheduling policy may use runtime information to recompute the dynamic priorities and ensure properties such as fairness of the selected messages; for instance, it may take advantage of the "aging" technique to avoid starvation.

3. TOOL ARCHITECTURE

We have implemented a tool to translate Creol programs into Java programs for execution, called *𝒞risp* (Creolized Interacting Scheduling Policies). 𝒞risp provides a one-to-one mapping from Creol classes and methods to their equivalent Java constructs. In order to implement active objects in Creol, we use the `java.util.concurrent` package (see Figure 1). Each active object consists of an instance of a process store and an execution manager to store and execute the method invocations.

Incoming messages to the active object are modeled as instances of `MethodInvocation`, a subclass of `java.util.concurrent.FutureTask` that wraps around the original method call. Therefore the caller can later get the result of the call. Additionally, `MethodInvocation` encapsulates information such as priorities assigned to the message.

The `ProcessStore` itself uses an implementation of the `BlockingQueue` interface in `java.util.concurrent` package. Implementations of `BlockingQueue` are thread-safe, i.e., all methods in this interface operate atomically using internal locks encapsulating the implementation details from the user.

The `ExecutionManager` component is responsible for selecting and executing a pending method invocation. It makes sure that only one method runs at a time, and takes care of processor release points.

In the following, we explain how the active object behaves in different scenarios, from a "client/server" perspective.

3.1 A New Method Invocation

A method call needs to be sent from the client to the server in an asynchronous way. To implement this in Java, the client first constructs an instance of `MethodInvocation` that wraps around the original method call for the server. Then, there are two implementation options how to add it to the server's process store:

ADD:

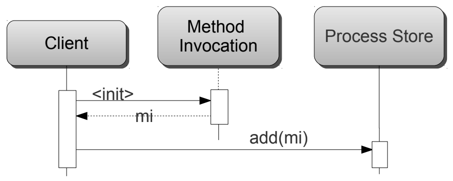

Figure 2: Adding the new MethodInvocations are performed on the Client side.

TAKE:

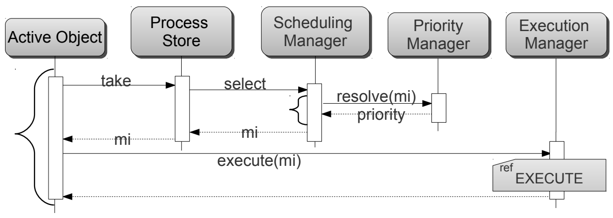

Figure 3: An active object selects a method invocation based on its local scheduling policy. After a method finishes execution, the whole scenario is repeated.

1. The client calls a method on the server to store the instance.
2. The client directly adds the method invocation into the process store of the server.

In option 1, the server may be busy doing something else. Therefore, in this case the client must wait until the server is free, which is against the asynchronous nature of communication. In Option 2, the Java implementation of each active object exposes its process store as an immutable public property. Thus, the actual code for adding the new method invocation instance is run in the execution thread of the client. We adopt the second approach as depicted in Figure 2. At any time, there can be concurrent clients that are storing instances of `MethodInvocation` into the server's process store, but since the process store implementation encapsulates the mechanisms for concurrency and data safety, the clients have no concern on data synchronization and concurrency issues such as mutual exclusion. The method or policy used to store the method invocation in the process store of the server is totally up to the server's process store implementation details.

3.2 Scheduling the Next Method Invocation

On the server side of the story, an active object repeatedly fetches an instance of method invocation from its process store for execution (cf. Fig 3). The process store uses its instance of `SchedulingManager` to choose one of the method invocations. 𝒞risp has some predefined scheduling policies that can be used as scheduling managers; nevertheless, new scheduling policies can be easily developed and customized based on the requirements by the user.

`SchedulingManager` is an interface the implementations of which introduce a function to *select* a method invocation based on different possible criteria (such as time or data) that is either predefined or customized by the user. The scheduler manager is a component used by process store when asked to remove and provide an instance of method invocation to be executed. Thus, the implementation of the scheduling manager is responsible how to choose one method invocation out of the ones currently stored in the process store of the active object. Different flavors of the scheduling manager may include time-based, data-centric, or a mixture.

Every method invocation may carry different levels of priority information, e.g., a server side priority assigned to the method or a client side priority. The `PriorityManager` provides a function to determine and resolve a final priority value in case there are different levels of priorities specified for a method invocation. Postponing the act of resolving priorities to this point rather than when inserting new processes to the store enables us to handle dynamic priorities.

3.3 Executing a Method Invocation

To handle processor release points, Creol processes should preserve their state through the time of awaiting. This is solved by assigning an instance of a Java thread to each method invocation. An `ExecutionManager` instance, therefore, employs a "thread pool" for execution of its method invocations. To create threads, it makes use of the factory pattern: `ThreadFactory` is an interface used by the execution manager to initiate a new thread when new resources are required. We cache and reuse the threads so that we can control and tune the performance of resource allocation.

When a method invocation has to release the processor, its thread must be suspended and, additionally, its continuation must be added to the process store. To have the continuation, the thread used for the method invocation should be preserved to hold the current state; otherwise the thread may be taken away and the continuation is lost. The original `wait` in Java does not provide a way to achieve this requirement. Therefore, we introduce `InterruptibleProcess` as an extension of `java.lang.Thread` to preserve the relation.

As shown in Figure 4, the thread factory creates threads of type `InterruptibleProcess`. The execution manager thread blocks as soon as it starts the interruptible process which executes the associated method invocation. If the method releases the processor before completion, it will be added back to the process store as explained in Section 3.1. When a suspended method invocation is resumed, the execution manager skips the creation of a new thread and reuses the one that was assigned to the method invocation before.

3.4 Extension Points

Besides the methods `add` and `take` for adding and removing method invocations, `ProcessStore` provides methods such as `preAdd` and `postAdd` along with `preTake` and `postTake` respectively to enable further customization of the behavior before/after adding or taking a method invocation to/from the store. These extension points enable the customization of priority or scheduling management of the method invocations.

𝒞risp provides two generic interfaces for priority specification and scheduling management: `PriorityManager` and `SchedulingManager` respectively. These two interface can be freely developed by the programmer to replace the generated code for priorities and scheduling of the messages. It will be the task of the programmer to configure the generated code to use the custom developed classes.

EXECUTE:

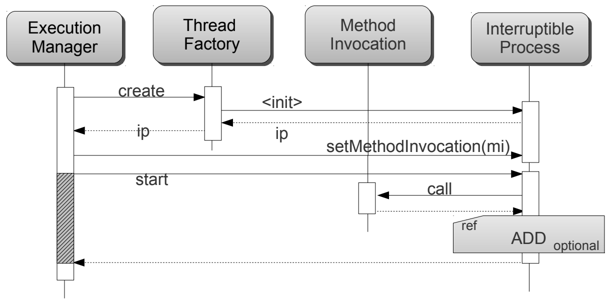

Figure 4: A method invocation is executed in an interruptible process. The execution manager thread is blocked while the interruptible process is running.

4. CASE STUDY

In this section, we demonstrate the use of application-level scheduling and 𝒞risp with a more complicated example: we program the "Sieve of Eratosthenes" to generate the prime numbers. To implement this algorithm, the `Sieve` is initialized by creating an instance of the `Prime` object representing the first prime number, i.e., `two`. The active behavior of `Sieve` consists of generating all natural numbers up to the given limit (100000 in our example) and passing them to the object `two`. A `Prime` object that cannot divide its input number passes it on to the next `Prime` object; if there is no next object, then the input number is the next prime and therefore a new object is created.

We parallelize this algorithm by creating active objects that run in parallel. The numbers are passed asynchronously as a parameter to the `divide` message. Correctness of the parallel algorithm essentially depends on the numbers being processed in increasing order. For example, if object `two` processes 9 before 3, it will erroneously treat 9 as a prime, because 3 is not there yet to eliminate 9. To avoid erroneous behavior, we use the actual parameter n in `divide` method to define its priority level, too (see line 15). As a result, every invocation of this method generates a process with a priority equal to its parameter. The default scheduling policy for objects always selects a process for execution that has the smallest priority value. This guarantees that the numbers sent to a `Prime` object are processed exactly in increasing order.

We used two different setups to execute the prime sieve program and compare the results. In one setting, we ran the parallel prime sieve compiled by 𝒞risp; in the other, we executed a sequential program developed based on the same algorithm that uses a single thread of execution in JVM. We performed the experiments on a hardware with 2 CPU's of each 2GHz with a main memory of size 2GB. We ran both programs for $max \in \{10000, 20000, 30000, 50000, 100000\}$.

The first interesting observation was that 𝒞risp prime sieve utilizes all the CPU's on the underlying hardware as much as possible during execution. This can be seen in Figure 6 which shows the CPU usage. Both CPUs are fairly in use while running this program. Figure 5 depicts the results of the monitoring of the parallel prime sieve in 𝒞risp using Visual VM tool. It depicts the number of threads generated for the program. This shows that 𝒞risp can handle a massive

Listing 2: Prime Sieve in Creol

```
interface IPrime begin
  op divide(n:Int)
end

class Sieve begin
  var n: Int, two: IPrime
  op init == two := new Prime(2); n := 3
  op run ==
    !two.divide(n);
    if n < 100000 then n := n + 1; !run() end
end

class Prime(p: Int) implements IPrime begin
  var next: IPrime
  op divide(n: Int) priority (n) ==
    if (n % p) ≠ 0 then
      if next ≠ null then
        !next.divide(n)
      else
        next := new Prime(n)
    end end
end
```

number of concurrent tasks.

One interesting feature of 𝒞risp is that the execution of any program under 𝒞risp can constantly utilize the *minimum* memory that can be allocated for each thread in JVM (thread stack). In JVM, the size of thread stack can be configured using `-Xss` option for every run. To demonstrate this feature of 𝒞risp, we collected the minimum stack size needed for every program run in Table 1. All 𝒞risp runs use the minimum thread stack size of `64k` that is possible for the JVM. On the contrary, the stack size required for the sequential version of the sieve program increases by the number of primes detected. This is also expected because of the long chain of method calls in the sequential sieve.

Having the constant thread stack size feature, 𝒞risp provides another interesting feature. It can handle huge number of thread generation if required. Table 2 summarizes the thread generation data for parallel prime sieve in 𝒞risp. It shows scalability of 𝒞risp as p rises for parallel prime sieve.

As the results show the use of Java threads is costly; first, 𝒞risp does not need much of the memory allocated to each thread and, second, the context switch cost is higher for larger memory allocation. In line with this, JVM uses a *one-to-one* mapping from an application-level Java thread to a native OS-level thread. In the current setting, the context switch of the threads are in the OS level. When the context switch is taken to the application level, we leverage the performance issue from the OS level to the application level. We further discuss this in Section 6.

5. RELATED WORK

The concurrency model of Creol objects, used in this paper, is derived from the Actor model enriched by synchronization mechanisms and coupled with strong typing. The Actor model [2] is a suitable ground for multi-core and distributed programming, as objects (actors) are inherently concurrent and autonomous entities with a single thread of execution which makes them a natural fit for distributed deployment [10]. Two successful examples of actor-based languages are Erlang and Scala.

max	10000	20000	30000	50000	100000
Sequential	64k	72k	96k	160k	190k
𝒞risp	64k	64k	64k	64k	64k

Table 1: Thread stack allocated for different executions

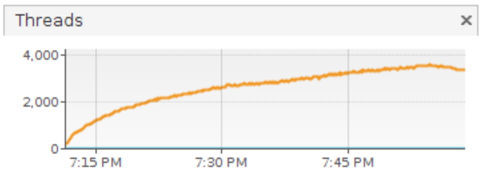

Figure 5: Increasing parallelism in 𝒞risp for Prime Sieve

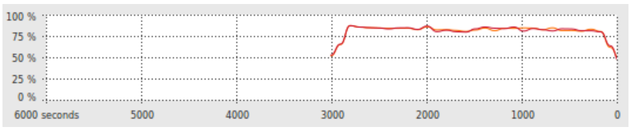

Figure 6: Utilizing both CPUs with Prime Sieve in 𝒞risp

Scala is a hybrid object-oriented and functional programming language inspired by Java. The most important concept introduced in [7, 1] is that Scala Actors unify *thread-based* and *event-based* programming model to fill the gap for concurrency programming. Through the event-based model, Scala also provides the notion of continutations. Scala provides quite the same features of scheduling of tasks as in concurrent Java; i.e. it does not provide a direct and customizable platform to manage and schedule priorities on messages corresponded among actors.

Erlang [4] is a dynamically typed functional language that was developed at Ericsson Computer Science Laboratory with telecommunication purposes [5]. Recent developments in the deployment of Erlang support the assignment of a scheduler to each processor [11] (instead of one global scheduler for the entire application). This is a crucial improvement in Erlang, because the massive number of light-weight processes in the asynchronous setting of Erlang turns scheduling into a serious bottleneck. However, the scheduling policies are not yet controllable by the application.

There are well-known efforts in Java to bring in the functionality of asynchronous message passing onto multicore including Killim [13], Jetlang [12], ActorFoundry [10], and SALSA [14] among others. In [10], the authors present a comparative analysis of actor-based frameworks for JVM platform. However, pertaining to the domain of priority scheduling of asynchronous messages, all provide a predetermined approach or a limited control over how message priority scheduling may be at the hand of the programmer.

In general, existing high-level languages provide the pro-

max	Live Peak	Total
10000	817	540591
20000	1468	1854067
30000	2204	4054814
50000	3707	11852985

Table 2: Number of live threads and total threads created for different runs of parallel prime sieve

grammer with little control over scheduling. The state of the art allows specifying priorities for threads or processes that are then used by the operating system to order them, e.g. Real-Time Specification for Java (RTSJ) and Erlang. In Crisp, we provide a fine-grain mechanism which allows for assigning priorities to high-level constructs, e.g., messages and methods.

Finally, we have considered, in previous work [6], local scheduling policies for Creol objects, with the purpose of schedulability analysis of real-time models. First of all, this paper is different as it investigates different levels of priorities that provide a high-level flexible mechanism to control scheduling. Secondly, we describe at present work how to compile Creol code to concurrent Java, and by allowing the use of class libraries in the underlying framework of Java, we can use Creol as a full-fledged programming language.

6. CONCLUSION

In this paper, we proposed $\mathcal{C}$risp as an implementation scheme for application-level scheduling of active objects. $\mathcal{C}$risp first introduces asynchronous message passing with fine-grain priority management and scheduling of messages. Additionally, it introduces a Creol to Java compiler that translates the active objects in Creol into an equivalent Java application. $\mathcal{C}$risp compiler seamlessly integrates Java class libraries into Creol including data types that turns Creol from a modeling language to a fully-fledged one in the hands of the programmer.

The `java.util.concurrent` package provides useful API for concurrent programming. Java futures facilitate modeling asynchronous message passing. However, for processor release points, we had to preserve threads (using `InterruptibleProcess`) to allow continuations which leads to their OS-level context switching that is costly. Moreover, we were tightly directed to use the out-of-the-box `ExecutorService` which is limitedly extensible. We had no control over the scheduling mechanisms of the internal queue used in the service implementations. Thus, we needed to re-implement some of the concepts. Through prototyping $\mathcal{C}$risp, we learned that there are two major challenges ahead. Firstly, we need to integrate continuations into Java using a many-messages-to-one-thread mapping model. Secondly, we need complete control over scheduling of messages and threads in `ExecutorService`'s internal queue. Table 3 summarizes this discussion.

	Asynchronous Communication	Processor Release Point	Scheduling
Modeling	✓	✓	×
Performance	✓	×	✓
	Java Futures	Interruptible Process	Executor Service

Table 3: Overview of evaluation of challenges

In future, we will focus on thread performance for $\mathcal{C}$risp such that thread scalability can be achieved to a certain limit. Additionally, the development of concurrency features on multi-core in $\mathcal{C}$risp is one of the major future concentrations. Moreover, another line of future work involves profiling and monitoring objects at runtime to be used for optimization and performance improvement. In addition, we intend to extend and integrate into our tool set model checking engines such as Modere [8].

7. REFERENCES

[1] *Coordination Models and Languages*, volume 4467, chapter Actors That Unify Threads and Events, pages 171–190. Springer Berlin / Heidelberg, 2007.

[2] Gul Agha. *Actors: a model of concurrent computation in distributed systems.* PhD thesis, MIT, 1986.

[3] G.R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, 2000.

[4] Joe Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, 2007.

[5] Fábio Corrêa. Actors in a new "highly parallel" world. In *Proc. Warm Up Workshop for ACM/IEEE ICSE 2010*, WUP '09, pages 21–24. ACM, 2009.

[6] Frank S. de Boer, Mohammad Mahdi Jaghoori, and Einar Broch Johnsen. Dating concurrent objects: Real-time modeling and schedulability analysis. In *CONCUR 2010*, volume 6269, pages 1–18. 2010.

[7] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2–3):202 – 220, 2009.

[8] Mohammad Mahdi Jaghoori, Ali Movaghar, and Marjan Sirjani. Modere: the model-checking engine of Rebeca. In *Proc. 21st ACM Symposium on Applied Computing*, pages 1810–1815, 2006.

[9] Einar Broch Johnsen and Olaf Owe. An Asynchronous Communication Model for Distributed Concurrent Objects. *Software and Systems Modeling*, 6(1):39–58, 2007.

[10] Rajesh K. Karmani, Amin Shali, and Gul Agha. Actor frameworks for the JVM platform: a comparative analysis. In *Proc. 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 11–20. ACM, 2009.

[11] K Lundin. Inside the Erlang VM, focusing on SMP. Presented at Erlang User Conference. Available at `http://www.erlang.se/euc/08/euc_smp.pdf`, November 13, 2008.

[12] Mike Rettig. *Jetlang Library*, 2008. `http://code.google.com/p/jetlang/`.

[13] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-Typed Actors for Java. In *ECOOP 2008 – Object-Oriented Programming*, volume 5142, pages 104–128. Springer Berlin / Heidelberg, 2008.

[14] Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with SALSA. *SIGPLAN Not.*, 36:20–34, December 2001.

[15] Ingrid Chieh Yu, Einar Broch Johnsen, and Olaf Owe. Type-safe runtime class upgrades in Creol. In *Proc. FMOODS'06*, volume 4037 of *LNCS*, pages 202–217. Springer-Verlag, June 2006.