Towards a light-weight approach for concurrent active objects in Java *

Charlie McDowell
University of California, Santa Cruz, USA
mcdowell@ucsc.edu

Olaf Owe
University of Oslo, Norway
olaf@ifi.uio.no

Abstract

The combination of object-orientation and concurrency can lead to severe programming difficulties. The Actor model has lately been used with success in object-oriented languages such as Scala, giving simpler synchronization and communication mechanisms. While Scala encourages functional style actor programming, the Creol language offers imperative style actor programming, and facilitates simple program reasoning. Verification of class invariants is similar to sequential style reasoning. In this paper we investigate an integration of this concurrency model in Java, providing a prototype implementation that facilitates Creol-style concurrent objects in Java.

Although Java currently provides classes to support concurrent objects, asynchronous method calls, and futures, it requires a very heavy weight framework, discouraging experimentation with the programming model. We are developing a very lightweight (from the programmer's perspective) approach to programming in this model using Java. In this paper we will describe our approach to programming concurrent objects in Java.

Categories and Subject Descriptors D.1.5 [*Object-oriented Programming*]; D.1.3 [*Concurrent Programming*]; D.3.3 [*Language Constructs and Features*]: Concurrent programming structures; F.1.2 [*Modes of Computation*]

Keywords Java, Creol, active objects, concurrent objects, asynchronous communication, futures, modular reasoning

1. Introduction

It is widely acknowledged that concurrency will play a role in an increasing percentage of programs written in the future as we hit the limits of Moore's Law. Multi-core processors are the norm, and the ever-growing demand for greater computation speeds can only be met with parallel computing. However, programming of modern concurrent systems is a complex and challenging task. The main paradigm for programming modern systems today is object-orientation, with Java-based languages as a common framework. For sequential systems the object-oriented, imperative setting offers a modular way of organizing programs. But this is easily compromised when non-trivial concurrency control statements are mixed with ordinary sequential application code in a non-modular way. Basic Java concurrency control mechanisms that compromise modularity include notification and signaling, synchronization on the same object from several classes, distributed locking and unlocking, shared variables through remote field access or missing synchronization. In particular these mechanism hinder modular understanding, verification and testing of class modules. Classes cannot be understood in isolation and reasoning about class invariants is extremely difficult. Progress, efficiency and deadlock control are also complicated.

The Actor model [1] has been proposed as a natural way to program concurrent systems. In the object-oriented world this can take the form of active concurrent objects that interact via asynchronous method calls. The concept of futures [5, 9] allows sharing of method results in a flexible manner. The Scala programming language unifies Java and the Actor model, avoiding the low-level synchronization mechanisms of Java. Scala Actors can be written in a functional way, with high-level support of futures (and promises) allowing non-blocking waiting for method results by means of call-backs from futures [6]. Scala suggests a programming style for actors that is less oriented towards classical imperative programming in favor of a functional programming style. Even though this may be seen as a strength of the language, it may be time consuming to learn by Java programmers.

Creol is an object-oriented programming model supporting the actor model by means of concurrent objects, but with a different philosophy on integration of functional and imperative styles than Scala. It uses a function sublanguage for programming of non-mutable data types, and uses imperative programming of (mutable) concurrent objects. The language avoids the Java synchronization mechanisms, has a modular formal semantics, and supports modular understanding and verification of classes [3, 4].

An actor in Creol is realized by a concurrent object, i.e., an object with a built-in thread performing calls on methods of that object and performing its own activity. Threads are not used at the programming level, and therefore the number of concurrent units is foreseeable, being the same as the

* This work was done in the context of the EU projects FP7-610582 *Envisage: Engineering Virtualized Services* (`http://www.envisage-project.eu`) and FP7-ICT-2013-X *UpScale: From Inherent Concurrency to Massive Parallelism through Type-based Optimizations* (`http://www.upscale-project.eu`).

number of created concurrent objects. Await statements allow suspension of method executions and allow non-blocking programming of asynchronous method calls (without callbacks as in Scala). This means that each concurrent object has its own process queue (of new or suspended method executions). This gives a simple imperative setting allowing modular verification of class invariants where reasoning inside a class resembles reasoning of sequential programs. Concurrent composition corresponds to logical conjunction of invariants, using communication histories to reason about communicated values.

Experimentation with alternative concurrency models might therefore be interesting. In order to encourage broader experimentation and development with the style of programming supported by Creol, we created a lightweight approach to "Creol-style" programming using standard Java. We call it lightweight because there is just one package containing two classes, CreolObject and CreolCall, and five methods that the user needs to know about.

Although Java's java.util.concurrent package provides support for Futures, those futures are not consistent with the active object concurrency model of Creol because the standard Java approach lacks the concept of an active object executing one process at a time. Each Java Future is connected to a separate runnable thread that will compute the future, but there is no implicit synchronization among runnable threads for the same object. Java also has support for asynchronous messages but only in the context of a complex framework of remote objects [7].

More importantly, programming with Java threads using synchronized, wait, notify, and notifyAll, is working at a very low-level of abstraction. Getting the synchronization "right" is very difficult, and debugging can be extremely difficult. Creol (and the Creol package in Java) provides a much higher level of abstraction. Using standard Java concurrency mechanisms, it is not possible to reason about the correctness of the code with regard to synchronization, by examining each class in isolation. The active objects of Creol can be analyzed independently.

The current implementation uses pure Java, not requiring any form of preprocessing. This simplicity does have a downside as discussed below. The addition of a preprocessor would eliminate some of these disadvantages as is planned as future work.

2. Creol

The key elements of Creol that we emulate in Java are futures, asynchronous method calls, active concurrent objects (executing at most one process/call at a time), and the ability to release control of an object via an **await** statement.

- **Fut**<T> `f;` declares `f` to be a future that will eventually contain a value of type `T`, i.e., `f` can be seen as a pointer to a global location where the future value eventually be stored. Futures are first-order in the sense that they can be passed as a parameter, and shared between objects.
- Execution of a method in an object is initiated with an asynchronous method call which takes the form `f = o!m(e);` where o is the object, m is the method, e is a parameter list, and `f` will hold the future, that will eventually be filled in by the method.
- The statement `x :=` **get** `f;` will block until the value for `f` has been produced. This blocks the entire object (which is only allowed to execute a single process at a time).

To allow for greater concurrency, Creol also provides two forms of **await** statements that release control of an object's processor.

- The statement **await** `x :=` **get** `f;` which suspends the current process (method execution) allowing the object to continue with some other method call or previously suspended call.
- The statement **await** `b;` where `b` is a Boolean expression will suspend the current process if `b` is false.

A process suspended via either form of the **await** will only be allowed to continue once it is enabled (the future produced or the Boolean expression becomes true) and the object's processor is released either at the completion of another method or when the active process suspends via an **await**. The syntax of Creol with futures is given in Figure 1 and includes standard statements such as if-statements, assignment (`v:=e`), and object creation (`v:=`**new** `C(`$\overline{e}$`)`). The ordinary blocking call, `v:=o.m(`$\overline{e}$`)`, can be seen as an abbreviation of `f:=o!m(`$\overline{e}$`); v:=`**get** `f` for some fresh future variable `f`. Similarly, non-blocking call, **await** `v:=o.m(`$\overline{e}$`)`, can be seen as an abbreviation of `f:=o!m(`$\overline{e}$`);` **await** `v:=`**get** `f`.

Creol Classes are reserved for definition of concurrent units, other data structures are defined by data types. We here include only standard types and lists, which suffices for the examples given. Class encapsulation is done by interfaces. One may only access an object through an interface. Therefore, an object variable must be declared by an interface (and not a class). This means that remote field access is forbidden, and the language does not support shared variables. We here consider a subset of Creol excluding inheritance, loop constructs, and the functional sub-language, which are not essential in our discussion.

A Creol example

Figure 2 reproduces a sample Creol program, taken from Din & Owe[2], that implements a publish/subscribe model. The program exemplifies non-trivial usage of futures. Clients may subscribe to a Service, which produces news messages to be sent to back to the subscribers of the service. A Service object uses proxies to handle all the subscribers. Each Proxy object is responsible for handling a fixed number of clients, and new proxies are created upon need (`nextProxy := new Proxy(limit,s)`). A Service object makes an asynchronous method call `fut := prod!detectNews()` to a Producer object `prod` responsible for generating another News value, which is then passed on to a proxy in the asynchronous method call `proxy!publish(fut)`. The server can continue without blocking or suspending, whereas the proxy blocks while waiting for the News value to appear in the statement `ns := get fut`, after which, it sends the future value to all its clients (`myClients!signal(ns)`) and makes another asynchronous `publish` call to the next proxy. When the last proxy is finishing its `publish` call, it makes an asynchronous call `s!produce()` back to the server `s` to initiate the next round of news generation.

In this program, we omit the final **put** statement of void methods. And an asynchronous method call may be done on a list of objects, as in the call `myClients!signal(ns)`, resulting in a multi-cast to each object in the list.

In	::=	**interface** I [**extends** I^+]$^?$ $\{S^*\}$	interface declaration
Cl	::=	**class** $C([T\ cp]^*)$ [**implements** I^+] $\{[T\ w\ [= e]^?]^*\ [s]^?\ M^*\}$	class definition
M	::=	$S\ B$	method definition
S	::=	$T\ m([T\ x]^*)$	method signature
B	::=	$\{[$**var** $[T\ x\ [= e]^?]^*;]^?\ [s;]^?$ **put** $e\}$	method blocks
T	::=	I | Void | Int | Bool | String | Fut$<T>$| List$<T>$	types
v	::=	$x \mid w$	variables (local or field)
e	::=	null | nil | this | v | cp | f($\overline{e}$)	pure expressions
s	::=	$v := e$ | **skip** | **if** e **then** s [**else** s]$^?$ **fi** | $s; s$	basic statements
		$\mid [fr :=]^? v!m(\overline{e}) \mid v := v.m(\overline{e}) \mid v :=$ **get** e	call-related statements
		| **await** $v := e.m(\overline{e})$ | **await** $v :=$ **get** e | **await** e	suspension
		$\mid v :=$ **new** $C(\overline{e})$	object creation

Figure 1. Syntax of core Creol, with C class name, cp formal class parameter, m method name, w fields, x method parameter or local variable, and where fr is a future variable. We let $[\,]^*$, $[\,]^+$ and $[\,]^?$ denote repeated, repeated at least once and optional parts, respectively, and $\overline{e}$ is a (possibly empty) expression list. Expressions e and functions f are side-effect free.

3. CreolJava

To create an active (Creol style) object using the *creol* package described in this paper, a class need only extend the class CreolObject. The functions provided by CreolJava are exemplified here.

- Make an asynchronous method call.
  ```
  fut = obj.invoke("methodName",
       param1, param2, ...);
  ```
- Block waiting for the future value to be produced.
  ```
  x = (TypeOfX)fut.get();
  ```
- Release the processor and wait for the future value to become available.
  ```
  x = (TypeOfX)creolAwait(fut);
  ```
- Release the processor. This method may resume immediately if there is no other process waiting.
  ```
  CreolSuspend();
  ```
- Release the processor. This method may resume only after some other activity occurs in the object.
  ```
  creolAwait();
  ```
  It is intended to be used with a while statement to effectively implement the Creol await on a condition. (See class `NewsProducer` line 8 in figure 5.)

The CreolJava version of the publish/subscribe example is shown in figures 3 and 5 (and will be contrasted with a version using standard Java synchronization below in figure 4). Line 13 in the `Service` class (fig. 3) shows a basic asynchronous call of the `detectNews()` method for the object `prod` from class `Producer`. Notice that method definitions are unaffected by being called asynchronously. The `detectNews()` method being called here asynchronously is defined in figure 5. Line 11 in class `Service` shows an example of a synchronous call (an asynchronous call combined with an immediate "get" of the value from the future). As discussed in the next section, when extracting a value from a future a cast is required.

Line 20 of class `Proxy` (fig. 3) uses `creolAwait()` to wait for a future to be produced, releasing the processor for this object to allow it to process other calls. Specifically here it will allow calls to `add()` to be processed while waiting for the `News` item to actually be produced. Once the news arrives, all of the clients managed by this proxy are asynchronously notified (line 21), and other proxies in this proxy chain are also notified asynchronously (line 23).

Line 6 of the `Producer` class (fig. 5) has an example of "getting" the value from a future, which blocks the processor for the object until the value for the future has been produced. Line 8 of class `NewsProducer` shows the stylized conditional await. The call `creolAwait()` will release the processor for the object, allowing other methods in this object to be processed. The suspended call will be re-enabled once some other method/process in this object has made some progress, potentially enabling the condition for this await. The `while` then re-checks the condition.

3.1 CreolJava vs Conventional Java

Figure 4 shows the Service, NewsProducer, and Client classes from an implementation of the original Creol publish/subscribe model using conventional Java synchronization and threads. To save space we have omitted the Proxy and Producer classes. The Proxy class is essentially identical to the CreolJava version except that the methods are both synchronized. The Producer class is unchanged except that is uses standard Java notation to invoke `np.getNews()`.

In this conventional Java version:

- The clients are each a thread that does something with news when it is received.
- The clients subscribe to a service.
- The service asks the Producer to detect news, and when it detects it, dispatches it to the Clients via a list of Proxies.
- The service is able to accept subscriptions at any time.
- The news items can be posted to the NewsProducer at any time.

The conventional Java version has less concurrency than the CreolJava version because the signal calls to the clients are processed sequentially by a single thread (the Service thread). In the CreolJava version the Proxies (one thread per Proxy) can quickly run down their respective chains firing off messages to the Clients without having to wait for any response from any Client. Also a number of news items could be queued up at the clients if they happen to arrive quickly.

Notice that in `Service`, the `subscribe` method must be synchronized (to avoid a race condition on the read/mod-

```
interface ServiceI {
    Void subscribe(ClientI cl);
    Void produce()}
interface ProxyI {
    ProxyI add(ClientI cl);
    Void publish(Fut<News> fut)}
interface ClientI {
    Void signal(News ns)}
interface ProducerI {
    News detectNews()}

class Service(Int limit) implements ServiceI {
    ProducerI prod; ProxyI proxy; ProxyI lastProxy;
    {prod := new Producer(); proxy := new Proxy(limit,this);
        lastProxy := proxy; this!produce()}
    Void subscribe(ClientI cl) {
        lastProxy := lastProxy.add(cl)}
    Void produce() {var Fut<News> fut;
       fut := prod!detectNews(); proxy!publish(fut)} }

class Proxy(Int limit, ServiceI s) implements ProxyI {
    List<ClientI> myClients = nil; ProxyI nextProxy;
    ProxyI add(ClientI cl) {
       var ProxyI lastProxy = this;
       if length(myClients)<limit
       then myClients := appendright(myClients, cl)
       else if nextProxy = null
       then nextProxy := new Proxy(limit,s) fi;
       lastProxy := nextProxy.add(cl) fi; put lastProxy}
    Void publish(Fut<News> fut) {var News ns = null;
       ns := get fut; myClients!signal(ns);
       if nextProxy = null then s!produce()
       else nextProxy!publish(fut) fi} }

class Producer(NewsProducerI np) implements ProducerI{
  News detectNews() {
    News news = null; news := np.getNews(); put news} }

class NewsProducer implements NewsProducerI {
  List<News> requests = nil;
  Void add(News ns) {requests := appendright(requests,ns)}
  News getNews() {
    var News firstNews = null;
    await requests /= nil;
    firstNews := head(requests);
    requests := tail(requests);
    put firstNews} }

class Client implements ClientI {
  News news = null;
  Void signal(News ns) {news := ns}}
```

Figure 2. The publish/subscribe example in Creol

ify of lastProxy) but the `produce` method must NOT be synchronized, otherwise the Service thread would be blocked waiting for news (detectNews) and unable to accept subscribe requests.

The NewsProducer must use notifyAll() in the add() method to resume any threads that were suspended waiting for news (in getNews). Naively using notify() here could result in some threads being left waiting when there is news available. As it turns out, in this program there will never be more than one thread (the Service thread) waiting inside of getNews() at any one time, so notify() would in fact work, however, this requires global reasoning about the program. It is also well documented that using synchronized methods rather than synchronized blocks also interferes with local reasoning because other classes could also synchronize on a NewsProducer object with a synchronized block issuing both wait() and notify() calls. This later could of course be avoided in our implementation by introducing a private local Object that is used as the lock rather than relying on the use of synchronized methods. The point here is that these are decisions and complexity that the programmer must deal with in Java, rather than concentrating on the core functionality of the system.

The Client class is clearly the most complex (relative to the CreolJava version). This is where the higher level of abstraction provided by CreolJava is most easily apparent.

3.2 Limitations of CreolJava

There are a number of disadvantages of this pure Java approach compared to Creol style asynchronous method calls. However, all of them can be addressed with the help of static analysis and a preprocessor.

As described above, asynchronous method invocation is made with the following syntax:

```
obj.invoke("methodName"[, parameter list])
```

As a consequence there are none of the usual compile time checks for proper spelling of the method name, or that there exists a method with the given name that accepts the offered parameter types. It would be possible to create a static analyzer that performs these checks. In the current implementation, any errors in the method name or parameter list will result in a NoSuchMethod exception at runtime. Another aspect of the method invocation process is that CreolObject methods must be public and CreolObject classes must also be public. This arises because the invocation using obj.invoke(...) is being made across package boundaries (from the package Creol to the application's package).

The result of CreolObject method calls is a future, represented by an instance of the class creol.Future. When extracting the contents of the future with `get()`, a cast is required.

```
1 Future fut = obj.invoke(''methodName''[, parameter list]);
2 SomeType val = (SomeType)fut.get();
```

The Java generic mechanism cannot be used here because different methods in the same CreolObject can return futures containing different types of values. Here also, a static analyzer could be built that would be able to automatically insert these casts.

This implementation also does not prevent a programmer from using the standard Java syntax for invoking CreolObject methods, potentially violating the "single active thread per object" assumption of the active object model. However, synchronous self calls are permitted using the standard Java syntax. The semantics of
`this.invoke("method"[,parameter list]).get()`
is the same as
`this.method([parameter list])` and for obvious reasons the latter is preferred. As with the previous limitations, this too can be checked with a static checker. The checker would allow standard syntax for CreolObject methods calls on *this*, but no others.

```
1  import creol.*;
2  public class Service extends CreolObject {
3    private Producer prod; private Proxy proxy;
4    private Proxy lastProxy;
5    Service(int limit, NewsProducer np) {
6      prod = new Producer(np);
7      proxy = new Proxy(limit, this);
8      lastProxy = proxy;
9      this.invoke(''produce''); }
10   public void subscribe(Client cl) {
11     lastProxy = (Proxy)lastProxy.invoke(''add'',cl).get(); }
12   public void produce() {
13     Future fut = prod.invoke(''detectNews'');
14     proxy.invoke(''publish'',fut); }}
```

```
1  import creol.*;
2  public class Client extends CreolObject{
3    private News news = new News(0);
4    public void signal(News ns) {
5      news = ns;
6      System.out.println(news); }}
```

Figure 3. The publish/subscribe example in Java using the *creol* package. (Part 1 of 2)

In Creol, the equivalent of non-Creol objects are always passed by value. In Java, these other non-Creol objects are passed by reference, resulting in multiple active CreolObjects potentially modifying the same data value. To properly embrace the Creol programming model, write access to shared objects (non-Creol objects) should not be allowed. This too can be detected using a static analyzer.

In summary, the features of a JavaCreol static checker should include:

- Perform the usual signature checking for methods invoked using the CreolObject invoke() method.
- Eliminate the need for the explicit type casts and ensure type safety for assignments involving get() to extract the contents of a future.
- Prevent the programmer from directly invoking CreolObject methods using standard Java method calling, except for self calls, which are then equivalent to synchronous self calls.
- Generate an error or warning for any shared write access to non-CreolObjects by CreolObject methods.

3.3 Implementation

The current version of the *creol* package is a modest 350 lines of code, including comments. The primary class for this lightweight active object implementation is the class CreolObject which extends Java's standard Thread class. The constructor for CreolObject automatically starts the thread which is the scheduler for the object, implemented in the run() method of CreolObject. A call to CreolObject.invoke() creates an instance of an internal class (CreolCall) that also extends Java's Thread class. When this call is scheduled to run by the object's scheduler (the run() method of the CreolObject), the CreolCall object uses reflection (java.lang.Class.getMethod()) and an instance of java.lang.reflect.Method to make the actual method call.

```
1  public class Service extends Thread {
2    private Producer prod; private Proxy proxy;
3    private Proxy lastProxy;
4    Service(int limit, NewsProducer np) {
5      prod = new Producer(np);
6      proxy = new Proxy(limit, this);
7      lastProxy = proxy; }
8    public void run() {
9      while(true) { produce(); }}
10   synchronized public void subscribe(Client cl) {
11       lastProxy = lastProxy.add(cl); }
12   public void produce() {
13       News news = prod.detectNews();
14       proxy.publish(news);}}
```

```
1  public class Client extends Thread{
2    private News news;
3    public void run() {
4      while(true) {
5        try {
6          synchronized(this) {
7            if (news != null) {
8              System.out.println(news); // consume it
9              news = null;
10             notify(); }
11           else {
12             wait(); }}}
13       catch (InterruptedException e) {}}}
14   synchronized public void signal(News ns) {
15     try {
16       if (news == null) {
17         news = ns;
18         notify(); }
19       else { wait(); }}
20     catch (InterruptedException e) {}}}
```

```
1  import java.util.*;
2  public class NewsProducer {
3    private ArrayList<News> requests =
4        new ArrayList<News>();
5    synchronized public void add(News ns) {
6      requests.add(ns);
7      notify(); }
8    synchronized public News getNews() {
9      try { while (requests.size() <= 0) wait(); }
10     catch (InterruptedException e) {}
11     News firstNews = requests.remove(0);
12     return firstNews; }}
```

Figure 4. The publish/subscribe example using standard Java threads and synchronization.

Calls to creolAwait() or creolSuspend() essentially invoke the scheduler for the object. Similarly, when a call terminates and produces a value for a Future, the scheduler for the object of the terminating method is invoked. In addition, any objects that might have been waiting for that future are also notified.

Each CreolObject maintains four queues: one for called methods that have not yet started, one for methods the voluntarily suspended (via creolSuspend()), one for methods suspended waiting for a future (creolAwait(future)), and one

```
1  import creol.*; import java.util.*;
2  public class Proxy extends CreolObject {
3    private int limit; private Service s;
4    private Proxy nextProxy;
5    private ArrayList<Client> myClients =
6      new ArrayList<Client>();
7    Proxy(int limit, Service s) {
8      this.limit = limit;
9      this.s = s; }
10   public Proxy add(Client cl) {
11     Proxy lastProxy = this;
12     if (myClients.size() < limit) { myClients.add(cl); }
13     else {
14       if (nextProxy == null) {
15         nextProxy = new Proxy(limit, s); }
16       lastProxy = nextProxy.add(cl); }
17     return lastProxy; }
18   public void publish(Future fut) {
19     News ns = (News)creolAwait(fut);
20     for (Client c : myClients) { c.invoke("signal",ns); }
21     if (nextProxy != null) {
22       nextProxy.invoke("publish",fut); }
23     else { s.invoke("produce"); }}}
```

```
1  import creol.*;
2  public class Producer extends CreolObject{
3    private NewsProducer np;
4    Producer(NewsProducer np) { this.np = np; }
5    public News detectNews () {
6      News news = (News)np.invoke("getNews").get();
7      return news; }}
```

```
1  import creol.*;
2  import java.util.*;
3  public class NewsProducer extends CreolObject{
4    private ArrayList<News> requests =
5      new ArrayList<News>();
6    public void add(News ns) { requests.add(ns); }
7    public News getNews() {
8      while (requests.size() <= 0) creolAwait();
9      News firstNews = requests.remove(0);
10     return firstNews; }}
```

Figure 5. The publish/subscribe example in Java using the *creol* package. (Part 2 of 2)

for methods suspended waiting for some condition other than a future (creolAwait()). Although not mandated by the Creol model semantics, each of these queues is managed in a first-come-first-serve manner. When a CreolObject is idle, the scheduler gives priority to called methods that have not yet started. If there are no new calls, then a method that voluntarily suspended is allowed to continue. Whenever the scheduler for an object activates a method to continue, any methods that suspended waiting from some condition using creolAwait() (other than waiting for a future) are moved from the conditional await queue to the voluntarily suspended queue. This will allow them to be scheduled at the next opportunity when it is assumed they will check the condition and either continue or wait again (see class `NewsProducer` line 8 in figure 5).

Each Future object also maintains a list of calls that have suspended waiting for the future. When the future is produced the Future object essentially transfers the waiting methods to the queue of voluntarily suspended methods. This gives an efficient implementation of futures.

4. Conclusion

We have taken the first step in creating a light-weight (from the programmer's perspective) implementation of Creol-style active objects using standard Java. There is no large infrastructure to setup, no new syntax, and just a handful of methods to learn. The interested reader, familiar with Java, can be up and running in just a few minutes. The source for the Creol package, a JavaDoc link, and the CreolJava example in this paper are available at `github.com/cmcwerner/creolJava`. All that is needed to get started is creol.jar from github.

A suit of tools has already been developed for Creol and its successor ABS [8], including compilers from Creol to Java. However, these tools are oriented towards programs written in Creol, which typically represent abstract program models. The present work allows Creol-style programming within Java itself, without the need to learn another language, and with all the facilities of Java available.

In the future we plan to develop a preprocessor and static analyzer that will eliminate the major drawbacks of this approach, without sacrificing any of the simplicity. The current implementation can take advantage of the parallelism of multi-core CPUs but does not have any mechanism for distributing objects over a network. It would be interesting to try and extend this "light-weight" approach with some support for distributed objects.

References

[1] G. A. Agha. *ACTORS: A Model of Concurrent Computations in Distributed Systems.* The MIT Press, Cambridge, Mass., 1986.

[2] C. C. Din and O. Owe. A sound and complete reasoning system for asynchronous communication with shared futures. *Journal of Logical and Algebraic Methods in Programming*, 83 (5-6):360–383, September 2014. .

[3] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.

[4] E. B. Johnsen, J. C. Blanchette, M. Kyas, and O. Owe. Intra-object versus inter-object: Concurrency and reasoning in Creol. *Electronic Notes in Theoretical Computer Science*, 243:89–103, July 2009. .

[5] B. H. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In D. S. Wise, editor, *Proc. SIGPLAN Conference on Programming Language Design and Implementation (PLDI'88)*, pages 260–267. ACM Press, June 1988.

[6] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala. A comprehensive step-by-step guide.* Artima Developer, 2008.

[7] Oracle. The Java EE 6 Tutorial: Overview of the JMS API, 2013. URL `http://docs.oracle.com/javaee/6/tutorial/doc`.

[8] The HATS project. The ABS tool suite, 2013. URL `http://tools.hats-project.eu`.

[9] A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86). Sigplan Notices*, 21(11):258–268, Nov. 1986.