A Proposal for Native Java Language Support for Handling Asynchronous Events

Carlos Rafael Gimenes das Neves[1], Eduardo Martins Guerra[1] and Clovis Torres Fernandes[1]

[1] Instituto Tecnológico de Aeronáutica – ITA, Computer Science Department, São Paulo, Brazil
{carlosrafael.prog, guerraem}@gmail.com, clovistf@uol.com.br

Abstract. During early stages of developing computer software, depending on the employed methodology, developers usually create the application's initial skeleton based on previously gathered requirements and on generated diagrams. When counting purely on what is provided by languages such as Java, developers tend to come up only with synchronous method calls, making use of coding tricks to achieve asynchronous behavior, which usually disrupts the original system model by adding a series of undesired side-effects such as unnecessary class coupling and error-prone constructions. This work proposed an extension to the Java language to allow for both executing asynchronous methods and handling asynchronous events occurring during normal execution, as a straightforward, class coupling-free and native alternative. With this extension it is expected that developers can natively use asynchronous communication from the beginning of the development cycle without having to make structural modifications to the original system.

Keywords: Asynchronous Event Handling, Language Structures, Compilers, Event Oriented Programming, Exception Handling

1 Introduction

Developing computer software is a task often performed with the aid of software methodologies and techniques that aim at providing developers with some level of ease and confidence. Along with that, code generation tools have recently become a more and more common choice for developers willing an extra level of confidence and convenience. In this scenario, asynchronous execution and communication is not a worry, as the tool would be in charge of it should it be necessary. Nonetheless, when only languages features can be used, considering the Java language, asynchronous execution and communication usually push developers towards solutions involving creation of new classes or interfaces not present in the original model, together with unplanned implementations and undesired class coupling, adding unnecessary complexity to both development and testing processes.

There is a series of commercial frameworks, for example Akka framework [1] and JMS API [2], deal with these complexities and provide developers with means to

achieve asynchronous execution and communication without too much effort. This topic is also the target of many academic works, such as JR Language, which proposes an extension to the Java language, making it a more suitable language for asynchronous communication and execution [3].

In spite of all the benefits brought by these works, some failed to deliver a transparent solution to developers [1-3], forcing them to change the original software model in some way, either implementing extra code or changing the original class hierarchy, in order to actually obtain the asynchronous behavior. On the other hand, others took care only of communication issues, even with first-class constructs, but left asynchronism aside [4]. Most of them also required the sender of an asynchronous message to know the receiver, creating a possibly undesired coupling between sender and receiver [1-3].

Our aim was to fill the gap between these works, proposing means for developers to naturally use asynchronous execution and communication, through first-class constructs and specially without forcing them to implement extra code, to couple parts of the software and to change the original software model of the software.

That behavior was achieved adding new language constructs to the Java language as well as creating a custom Java compiler capable of interpreting them. These new language constructs enable developers to easily add specific asynchronous behavior to the system without side-effects, as they act just as native structures do. Moreover, the fact that neither extra code nor changes to the class hierarchy are required makes the resulting code more compliant to the original model, improving both code readability and faithfulness to the documentation where the system derives from.

2 Characterizing the Research Problem

A close inspection to a typical Java program shows that the native structures available for communication fall basically into two groups, namely, method calls and thrown exceptions [5], [6]. Both are first-class constructs that work with the call stack and present a synchronous behavior. Considering method calls and thrown exceptions as messages, the execution actually relies on first-class constructs for synchronous message passing between methods in the call stack. Method calls differ from throwing exceptions, as the throwing of an exception makes the sender method terminate [5].

Existing tools and works regarding message-passing techniques present different approaches. Some provide asynchronous behavior, but force changes to the structure of existing classes, or require implementation of extra code [1-3]. Some come as first-class constructs, but without the asynchronous behavior [4]. Coupling of the sender and the receiver is also present [1-3]. Access to the original call stack is also provided only by some of them [3], [4], [11], just as access to the original context [4], [5], [11].

Therefore, the main problem addressed in this work is how to allow code to execute asynchronously through first-class constructs, giving it the ability to both asynchronously notify other methods in the call stack about the occurrence of general events and access the original context, without the need for interrupting the execution of the sender method, for coupling classes or for implementing extra interfaces.

3 Proposed Solution

One possible solution to the issue previously brought up would be creating extensions to the Java language, in order to provide first-class constructs for developers to handle asynchronous events, without implementing extra interfaces and without worrying about usual matters on asynchronous event handling, such as registration control and capture of the method's context, namely, its closure [7].

Unlike the capture performed by Java up to version 1.7, which included only local variables marked as final [5], [6], which are immutable, the captured closure must be mutable. Moreover, it must be accessible from different threads at the same time, as the asynchronous method execution will be accomplished through the creation of several concurrent threads.

Since in Java, a method's closure is composed of both local variables and parameters, capturing the closure and making it accessible from another thread means giving this other thread access to the memory area where that information is stored. According to the Java Virtual Machine specification [6], the memory is arranged similarly to the diagram in Fig. 1, with all local variables stored inside frames.

New frames are created every time a method is called, and placed at the top of the stack. Frames are destroyed and thus, lost, when its owner method terminates [6]. No method, from any thread, can access frames directly, either before or after the frame's destruction.

The only way to access frames during runtime is through tools such as the Java Virtual Machine Tool Interface or the Java Debug Interface, both part of the Java Platform Debugger Architecture [8]. Nevertheless, there are three important reasons to avoid using these tools in the solution. First, there is no guarantee that these tools

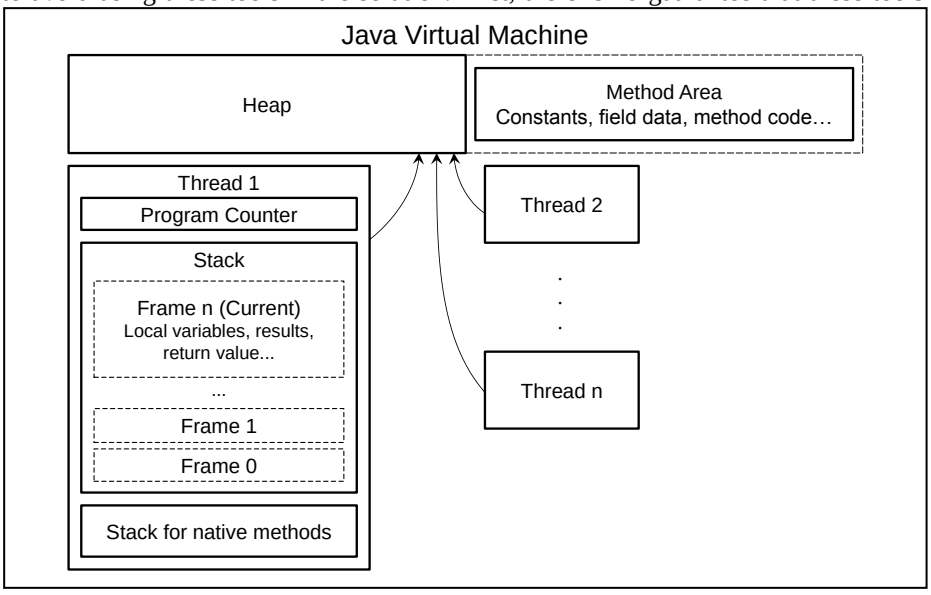

Fig. 1. Simplified internal memory distribution of the Java Virtual Machine

will be part of all platforms where Java is supported. Also, the thread must be in a suspended state to have its frames available for inspection. Last, using those tools involves writing some native code, removing the portability of the solution.

For the solution to be portable it should only make use of native Java commands and structures. Since the traditional Java compiler is capable of capturing only local variables marked as final, in order to capture all local variables while keeping then mutable, the solution would be to move them into a local class, similar to what has been done in [9]. This local class has one single local instance marked as final, capable of being captured by the traditional Java compiler. Since parameters cannot be moved, they are actually copied into the local class.

Then, all their references are replaced with references to the fields of this newly created class, as shown in Fig. 2. As a consequence of those changes, inline variable initializers must be moved to their own line. Variables and parameters that are marked as final can be left just as they are, since traditional Java compilers can already capture them.

Original Method

```
String m1(int p) {
  final String f =
    "...";
  String s =
    p.toString();
  s += f;
  return s;
}
```

Modified Method

```
String m1(int p) {
  class Closure {
    int p;
    String s;
    Closure(int p) {
      this.p = p;
    }
  }
  final Closure c = new Closure(p);
  final String f = "...";
  c.s = c.p.toString();
  c.s += f;
  return c.s;
}
```

Fig. 2. Code modification to allow the method's closure to be properly captured

Although that technique does enable the closure to be captured, it is just a part of the solution being proposed. In order to actually solve the problem, this technique must be performed automatically, while still allowing methods in the call stack to communicate asynchronously through first-class constructs, which involves adding new language constructs to the Java language. There are many tools available to help in the creation of extended Java compilers, capable of generating standard Java Bytecode, executable in any standard Java Virtual Machine. Among all the available tools, JastAddJ [10] was chosen due to its relative simplicity and powerful reference attribute grammars.

Using JastAddJ, seven new language constructs were added to the Java language. They form structures that are translated into ordinary Java code referencing existing precompiled helper classes that are responsible for simulating the expected behavior.

One language construct, *within*, is used to specify regions in the code where to monitor the occurrence of events, similar to what *try* statements do in plain Java. It is used together with two other language constructs, *when* and *xwhen*, which specify the code that is to be asynchronously executed when a particular event happens inside the region delimited by *within*. The difference between *when* and *xwhen* is that the latter serializes simultaneous executions while the former allows as many concurrent executions as necessary. Both language constructs are analogous to the *catch* clause of the Java language. The third language construct added was *event*, used for asynchronous event notification, similar to what *throw* statements do in plain Java with exceptions. With these four language constructs, developers are able to produce general event notifications, to specify the region where events are monitored and to asynchronously handle these events, using first-class constructs without implementing extra interfaces or dealing with any kind of registration control.

Another language construct, *arun*, was added to provide developers with means to run ordinary asynchronous code unconditionally, while still allowing events occurred during this asynchronous execution to be properly handled by previous *when* and *xwhen* blocks. The last two constructs, *handlerStack* and *currentHandlerStack*, were added to allow event notifications to be properly handled in scenarios where code must be run asynchronously but *arun* cannot be used, such as when new threads must be explicitly created by the developer.

3.1 *when, within* and *event* Constructs

The *when*, *within* and *event* language constructs constitute the foundation of the proposed solution. A *within* statement behaves similarly to what an ordinary *try* statement does [5]. It is used to determine the region where to monitor the occurrence of events. Any methods called in the *within* block are also monitored, as well as the methods called inside those methods and so on. One or more *when* clauses must precede a *within* statement to specify the behavior for each event that the developer wishes to monitor the occurrences.

A *when* clause behaves like an ordinary *catch* clause [5], specifying what code to execute upon the occurrence of an event of a particular class. Unlike the *catch* clause that can be used only with the class *Throwable* or with its subclasses [5], there are no limitations to what classes can be used with a *when* clause giving developers greater flexibility and freeing them from changing the current class hierarchy in the system, just to be able to use that class as an event. Code inside *when* blocks always executes asynchronously. Also, if an event of class A is expected, but an event of class B is triggered, and B extends A, the *when* block expecting the event of type A is executed.

Finally, the *event* statement is used like an ordinary *throw* statement, notifying the occurrence of an event. The main difference is that *event* statements do not interrupt or abort current method's execution, as they execute asynchronously. One single event can trigger multiple *when* blocks, because unlike usual Java exceptions, that are propagated only until the first handler is found [5], events were defined to propagate to all suitable, currently active *when* handlers.

Actual exceptions thrown in a *within* block are not automatically handled, and should be handled manually by the developer. On the other hand, in order to address the issues of handling exceptions in asynchronous environments [11], it has been stipulated that if an exception is thrown during the asynchronous execution of a *when* block, and the developer does not catch that exception inside the block, the exception is transformed into an this event is notified so that it could be handled by other *when* blocks expecting that exception or one of the exception's super-classes.

3.2 *xwhen* Construct

We can use the *xwhen* clause for replacing a *when* clause, to indicate that no more than one simultaneous execution of that block can exist. When an event triggers a *when* block and that block is already executing, another execution of that block starts. On the other hand, when an event triggers an *xwhen* block and that block is already executing, the notification is queued, and the block executes again only after having finished the present execution.

3.3 *arun* Construct

The *arun* statement is used to asynchronously execute a block of code. What differs using *arun* and running another thread is that events occurring inside the *arun* block are propagated to the handlers present in the stack, while a new ordinary thread will have an empty stack of handlers.

3.4 *handlerStack* and *currentHandlerStack* Constructs

Since all event notifications are propagated to the current stack, should developers need to manually create a new thread, in situations where *arun* cannot be used, that new thread will have a new blank stack of its own, and any events occurred there will not be propagated to the stack of the method creating that thread. The new primitive data type *handlerStack* and the statement *currentHandlerStack* have been created for this kind of situation. The *currentHandlerStack* statement allows developers to obtain a copy of the current stack of handlers and to use that copy as the target of an event notification or as the initial stack used for an execution of an *arun* block. The new data type *handlerStack* is the data type of the copies of the current stack.

4 Implementation Details

As the solution introduces seven new constructs to the Java language, it is necessary to demonstrate what happens behind those constructs, specially the aspects regarding their inner workings, such as how they are converted into normal Java code and how they interact with the precompiled helper classes. Although not all these helper classes are explained in this work for brevity, the behavior of the main class referenced by the transformed code, the *Manager* class, is explained with some detail.

4.1 *when, within* and *event* Constructs

After having defined the behavior of *when*, *within* and *event* constructs, the following code snippet helps clarify their basic usage.

```
class A {
  void m1() {
    when (Event e) {
      m3(); //Code to handle events of class Event
    } when (Event2 e2) {
      m4(); //Code to handle events of class Event2
    } within {
      m2();
    }
  }
  void m2() {
    event new Event();
    m5();
  }
  void m3() { event new Event2(); }
  void m4() {...}
  void m5() {...}
}
```

[Sample code demonstrating the usage of *within*, *when* and *event* constructs]

As it can be seen in the diagram illustrated in Fig. 3, the eventual notification of the event of type *Event* in *m2( )* triggers one of the *when* blocks inside *m1( )*; in fact, the first *when* block inside *m1( )*, which includes only a call to the method *m3( )*. Therefore, that block is executed asynchronously on a new thread, even though *m1( )* is not the current method on the call stack of the thread where the notification occurs; the current method is *m2( )*. By the time *m3( )* executes and eventually notifies about the occurrence of event of type *Event2*, there is no guarantee whether *m1( )* will still be in its corresponding call stack, which does not matter, as the *when* block for *Event2* is triggered and method *m4( )* is executed even if *m1( )* has already terminated.

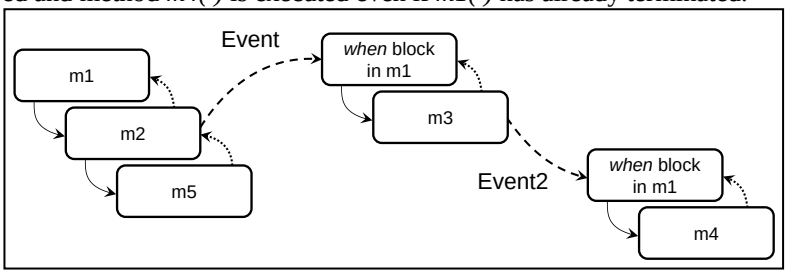

Fig. 3. Block diagram showing the execution of the sample code using *within*, *when* and *event*

Events occurring in one *when* block of a given method are seen as if occurring in its companion *within* block, below it. Consequently, these events can trigger any *when* blocks above that *within* block, as well as other when blocks that were registered in the stack at the time the *within* block started executing. This behavior can be better understood by looking at the following code.

```
class A {
  void m1() {
    when (Event e) { m4(); }
    when (Event2 e) { m6(); }
    within { m2(); }
  }
  void m2() {
    when (Event e) { m5(); }
    when (Event2 e) { m7(); }
    within { m3(); }
  }
  void m3() { event new Event(); }
  void m4() {...}
  void m5() { event new Event2(); }
  void m6() {...}
  void m7() {...}
}
```

[Sample code demonstrating the behavior of *within* and *when* in more detail]

Assuming that the execution starts with a call to method *m1( )*, method *m2( )* is the next one to be called, followed by a call to method *m3( )*, which eventually notifies about the occurrence of the event of type *Event*. That notification triggers the *when* blocks in *m1( )* and in *m2( )*, and the methods *m4( )* and *m5( )* are called. Once *m5( )* is called, the event of type *Event2* ultimately occurs, triggering the second *when* blocks in *m1( )* and in *m2( )*, finally causing methods *m6( )* and *m7( )* to be called.

Again, all this behavior is thanks to the precompiled helper classes referenced by the transformed code, presented in Fig. 4. The transformation does not actually change the source files, it just changes the abstract syntax tree generated by JastAddJ before the code is actually compiled [10].

4.2 *xwhen* Construct

As already explained, the behavior of *xwhen* is different in comparison with the behavior of *when* clauses. In spite of that, there are not too many differences in the transformed code, except for the first parameter of the method *append( )*, shown in Fig. 4, which is changed from *true* to *false*, telling the helper class *Manager* to change the execution behavior during runtime.

Original Code	Transformed Code

`void m1(int p) {`
`  int a = 0;`
`  when (Event e) {`
`  a += p;`
`  } within {`
`  a = 10;`
`  m2();`
`  }`
`}`
`void m2() {`
`  event new Event();`
`}`
	`void m1(int p) {`
	`  class Closure {`
	`  int p, a;`
	`  Closure(int p) { this.p = p; }`
	`  }`
	`  final Closure c = new Closure(p);`
	`  c.a = 0;`
	`  Manager.startAppending();`
	`  try {`
	`  Manager.append(true, Event.class,`
	`  new Handler<Event>() {`
	`  public void handle(Event e) {`
	`  c.a += c.p;`
	`  }});`
	`  Manager.endAppending();`
	`  c.a = 10;`
	`  m2();`
	`  } finally { Manager.rewind(); }`
	`}`
	`void m2() { Manager.notify(new Event()); }`

Fig. 4. Transformation applied to the code before compiling *when*, *within* and *event* constructs

4.3 *arun* Construct

The following snippet helps demonstrating the runtime behavior of *arun* blocks, as well as Fig. 5 shows what *arun* blocks look like after the code transformation.

```
class A {
  void m1() {
    when (Event e) { m5(); }
    within { m2(); }
  }
  void m2() {
    arun { m3(); }
    m5();
  }
  void m3() { event new Event(); }
  void m4() { ... }
  void m5() { ... }
}
```

[Sample code demonstrating the behavior of *arun* blocks]

Assuming that the execution starts with a call to method *m1( )*, method *m2( )* is the next one to be called. There, the *arun* block is executed asynchronously, and the method *m3( )* is called from another thread, which notifies about the occurrence of the event of type *Event*. That notification triggers the *when* block in *m1( )* and the method *m5( )* is called.

Original Code	Transformed Code

`void m1(int p) {`
`  arun {`
`  int a = p + 1;`
`  m2(a);`
`  }`
`}`
`void m2(int x) {...}`
	`void m1(int p) {`
	`  class Closure {`
	`  int p;`
	`  Closure(int p) { this.p = p; }`
	`  }`
	`  final Closure c = new Closure(p);`
	`  Manager.fork(new Runnable() {`
	`  public void run() {`
	`  int a = c.p + 1;`
	`  m2(a);`
	`  }});`
	`}`
	`void m2(int x) {...}`

Fig. 5. Transformation applied to the code before compiling the *arun* construct

When an actual exception is thrown during the asynchronous execution of an *arun* block, it is caught and transformed into an event that could be handled by a when block expecting that exception or one of its super-classes.

4.4 *handlerStack* and *currentHandlerStack* Constructs

The statement *currentHandlerStack* is translated into a simple call to a method of the Manager class which returns a copy of the current stack. The data type *handlerStack* is translated into a reference to a normal interface. The basic usage of *handlerStack* and *currentHandlerStack* constructs is demonstrated in the following code snippet.

```
handlerStack cs = currentHandlerStack;
event new Event() cs;
arun cs { ... }
```

[Sample code demonstrating the usage of *handlerStack* and *currentHandlerStack*]

Once a copy of the stack is made, it can be used together with the *event* statement, forcing the event to be notified in the given stack, or it can be used with *arun*, forcing the given stack as the initial one used by *arun*.

4.5 Precompiled Classes

The code translation itself is not enough to give the desired behavior to the solution, as this requires some code to be executed. All this code necessary to bring life to the solution was encapsulated in precompiled helper classes. Instead of referencing many classes, it was decided to create the *Manager* class, and make it the only class referenced by the transformed code, apart from the interface referenced by the *handlerStack* data type.

The *Manager* class acts as a façade (kind of design pattern [12]) to the remaining precompiled helper classes, calling their methods as necessary. As this helper class is in charge of making the solution actually work during runtime, next we present an overview of its methods called in the transformed code shown in Fig. 4 and Fig. 5:

- startAppending( ) – It creates an empty list of handlers and adds this list to the top of an internal stack of lists, which is maintained on a per thread basis using the concept of thread-locals variables [13].
- append( ) – It appends the handler to the list on the top of the current stack, specifying the class of the event that the handler is prepared for.
- endAppending( ) – It creates a copy of the current internal stack, propagating that copy to all handlers in the list at the top of the stack, so that events occurring during the asynchronous execution of one of those handlers can be properly notified to other handlers in the current stack.
- rewind( ) – It removes from the stack the list of handlers at the top of it.
- notify( ) – It notifies all suitable handlers in the internal stack about the occurrence of an event, starting their asynchronous execution.

5 Discussion

Five discrete topics come out by dividing the problem discussed in this work: asynchronous method execution, asynchronous event notification and handling, class coupling, implementation of extra interfaces and ability to access the original call stack and context. All these five topics were addressed by the implementation previously explained.

Asynchronous method execution is achieved through the *arun* first-class construct. It allows not only for methods to be executed asynchronously but also for any kind of structured code to be executed asynchronously. Once the *arun* block is executed it creates another thread, without developer's interaction, to execute its contents and the code following the *arun* block continues as soon as this new thread starts.

With *event* statements the developer can asynchronously notify about the occurrence of events as these statements do not block the execution of the current method. On the other end, with *when* blocks developers can asynchronously handle these events. The execution of one *when* block is not tied to the execution of any other *when* block, therefore, many *when* blocks can be executing at the same time.

Class coupling is not a concern as the sender of an event communicates with all the handlers present in the call stack, independently of their class. Also, the handlers need

not to know which class is responsible for generating an event, they should only care about the area of code where the desired event could happen.

Implementation of extra interfaces is also not a concern for the developer, since all constructs provided by the solution are first-class constructs placed throughout the code along with other ordinary native structures, they do not require any changes to be applied to the declaration of the classes.

The access to the original call stack is guaranteed by the automatic registration technique, shown in Fig. 4, which creates lists of registered handlers on a per thread basis. These lists are propagated to the new threads created by *arun* and *when* blocks, in such a way that the events triggered within those threads can be propagated to the original handlers. Last, the access to the original context is accomplished through the code transformation shown in Fig. 2 and Fig. 4, which encapsulates all local variables and parameters inside a local class, making them accessible from within the handlers.

6 Related Works and Tools

Asynchronous communication and asynchronous method execution is a fascinating broad topic. As such, it is issued by several academic works and commercial tools, some of which are discussed next.

6.1 Akka Framework

Akka [1] is a commercial Java framework focused on parallel and distributed computing, helping in the creation of highly scalable and fault tolerant applications, offering many features that extend beyond the scope covered by this work. The main interest in Akka is its message passing functionality. In a brief description, it is based on the concept of message passing between actors. Actors are the components that actually execute code in the system, similar but not equal to what classes are to object oriented programming, as it is possible to assign a different class to the role of one specific actor at different times [1].

Each actor has one mailbox where the messages are kept until they are processed and they do not need to reside on the same process in order to communicate. Moreover, their distribution can be hierarchical in such a way an actor can create other actors, delegate them tasks and supervise them [1].

Considering the sending of a message as an event and the handling of the message as the handling of the event, Akka provides the developer with asynchronous event handling and with many other features not covered by this work, such as asynchronous execution and communication in physically distributed environments. Albeit there is no need for the classes to know each other, they must know the actor they want to send a message to, which can be seen as a type of coupling. Also, the framework requires classes to extend one of the *Actor* classes provided in order to play the role of an actor, changing the original structure of the system. Last, because it is an external framework, upon which software can be developed, Akka does not behave as a first-class construct.

6.2 Spring Framework / JMS

Spring [14] is a commercial modular framework for developing software applications, which integrates with integrates with the Java Messaging System API, JMS API [2], in order to provide an asynchronous and flexible messaging system. JMS works based on two distinct models, Point-To-Point and Publish/Subscribe [2]. In the Point-To-Point model there are message queues acting as mailboxes, where messages are sent to and retrieved from. In the Publish/Subscribe model, everyone interested in one topic subscribe to that topic, so they can receive all messages that are published under that topic. Since both message queues and topics implement one special interface, called Destination [2], the difference between the models becomes transparent for those accessing only the Destination interface.

Considering the sending of a message as an event and the handling of the message as the handling of the event, Spring + JMS provides the developer with asynchronous event handling. Actually, the integration Spring + JMS offers a highly customizable platform for application development in many aspects that surpass the scope of this work. In spite of this fact, for the asynchronous event handling to work properly, the sender of the message must know the message's destination, coupling sender and receiver. Yet, neither the sender nor the receiver make use of first-class constructs and any classes willing to receive messages must implement one extra interface in order to be able to do so.

6.3 JR Language

The JR language [3] is actually an extension to the Java language, adapted from the SR language [15], [16], aiming at providing better Java support to distributed and concurrent computing. As a part of that effort, the extension provides several features such as remote object creation, remote method invocation, and asynchronous message passing, among others. Its asynchronous message-passing model is based on new constructs introduced by the language, specially *send* and *forward*.

The general format of the commands is as follows:

```
send target.message(arguments) handler handlerObject;
forward target.message(arguments) handler handlerObject;
```

[General format of the *send* and *forward* commands in the JR language]

Both *send* and *forward* commands send the given message to the target. But the *forward* command additionally delegates to the given method the obligation to return a value to its caller. The *handler* part of the command is used to indicate the object responsible for handling any exceptions that may occur during the asynchronous execution of the given method. In order to do so, that object must implement one handler method for each possible exception.

Thereby, the JR language works with two different models of event handling, one for the exception handling and another one for the explicit messages. Its exception handling mechanism behaves synchronously and is similar to the native Java mecha-

nism, with the difference that it relies on methods to handle the exceptions, not on *try-catch* clauses. In contrast, its asynchronous message-passing model provides a straightforward mechanism that uses first-class constructs and does not require the implementation of any extra interfaces. Yet, it still requires the sender of the message to know the receiver, coupling both classes.

6.4 Robust Exception Handling in an Asynchronous Environment / ProActive

Robust exception handling in an asynchronous environment proposes transparent means for handling exceptions occurred in asynchronous method calls [11], using regular Java constructs together with the tool ProActive [17]. The main idea is to actually throw the exception generated by the asynchronous method, if any, upon the first use of the object returned by the asynchronous method, similarly to what Java's Future objects [18] do, but in a transparent fashion. If the object returned is not used within the *try* block, the exception is thrown at the end of the *try* block [11].

Similarly to the JR language, this work has two different models of event handling, one for the explicit messages, which is similar to the JR language model, and another for the exception handling, which, in turn, is analogous to the native Java one. As the explicit asynchronous message-passing model is analogous to the proposed by the JR language, it does provide means to communicate asynchronously using first-class constructs and without requiring the implementation of any extra interfaces, at the same time it still requires the sender to know the receiver, coupling both classes.

6.5 Exception Handling with Resumption: Design and Implementation in Java

Exception handling with resumption shows an alternative for the native Java exception handling model [4]. In this proposed model the method where the exception occurs is not necessarily terminated. After the occurrence of the exception, the handler has a chance to try to solve the problem and propose a solution. If the solution is accepted by the originating method, then its execution continues. This behavior is achieved through the use of two new language constructs, *resume*, used within a catch block to try to propose a solution, and *accept*, used with *throw* clauses, to specify acceptable solutions for that particular exception.

Although this work proposes a way to prevent terminating abruptly the execution of a method using first-class constructs and without any extra interfaces, its execution does get interrupted momentarily, as there is no kind of asynchronous behavior. Also, its entire notification model is to be used with exceptional conditions only, not with general events.

7 Conclusion

Asynchronous execution and event notification are truly fascinating topics that can be looked upon from different perspectives. This work proposes a slightly different form of viewing and dealing with those topics, focusing on asynchronous communication between methods in the call stack as well as their asynchronous execution.

While the proposed solution does provide developers with clean, native means for methods to communicate asynchronously, without requiring them to implement extra interfaces, create unnecessary class coupling and control the registration of active event handlers manually, the final code that is actually executed adds several new classes to the program and makes extensive use of handler registration control, but as an automatic process. Even so, the final code neither changes the original classes' hierarchy nor creates unnecessary coupling between them.

Although the presented solution offers enough features to address the aforementioned problem, there is still plenty of room for future improvements and future works such as creating a *multi-when* version of the *multi-catch* present in the Java language [5] or even providing the developer with means for manually creating threads with custom-initialized stacks. Another possible improvement would be providing developers with means to help with synchronization issues and automatic mutual exclusion control of shared resources used inside *when* and *arun* blocks.

References

1. Typesafe Inc.: Akka Documentation 2.0. http://akka.io/docs/ (2011)
2. Sun Microsystems: Java Message Service Specification 1.1. http://download.oracle.com/otndocs/jcp/7195-jms-1.1-fr-spec-oth-JSpec/ (2002)
3. Keen, A. W., Ge, T., Maris, J. T., Olsson, R. A.: JR: Flexible distributed programming in an extended Java. In: ACM Transactions on Programming Languages and Systems, pp. 578-608. ACM, New York (2004)
4. Gruler, A., Heinlein, C.: Exception handling with resumption: design and implementation in Java. In: Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC'05), pp. 165-171. Las Vegas (2005)
5. Gosling, J., Joy, B., Steel, G., Bracha, G., Buckley, A.: The Java Language Specification, Java SE 7 Edition. http://docs.oracle.com/javase/specs/ (2011)
6. Lindholm, T., Yellin, F., Bracha, G., Buckley, A.: The Java Virtual Machine Specification, Java SE 7 Edition. http://docs.oracle.com/javase/specs/ (2011)
7. Krishnamurthi, S.: Programming Languages: Application and Interpretation. http://www.cs.brown.edu/~sk/Publications/Books/ProgLangs/ (2007)
8. Oracle: Java Platform Debugger Architecture (JPDA). http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/ (2012)
9. Heinlein, C.: Local Virtual Functions. In: Proceedings of NODe 2005, GSEM 2005, pp. 129-144. Erfurt (2005)
10. Ekman, T., Hedin, G.: The JastAdd Extensible Java Compiler. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 1-18. Montreal (2007)

11. Caromel, D., Chazarain, G.: Robust exception handling in an asynchronous environment. In ECOOP Workshop on Exception Handling in Object-Oriented Systems: Developing Systems that Handle Exceptions, number 05050 in Technical Reports - Laboratoire. Sophia Antipolis (2005)
12. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing, Boston (1995)
13. Oracle: ThreadLocal Class Documentation. http://docs.oracle.com/javase/7/docs/api/java/lang/ThreadLocal.html (2012)
14. Spring Source: Spring Framework 3.1. http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/pdf/spring-framework-reference.pdf (2012)
15. Olsson, R. A., Andrews, G. R., Coffin, M. H., Townsend, G. M.: SR: A Language for Parallel and Distributed Programming. The University of Arizona, Tucson (1992)
16. Andrews, G. R., Olsson, R. A.: The SR programming language: concurrency in practice. Benjamin-Cummings Publishing, Redwood City (1993)
17. ActiveEon: ProActive Parallel Suite. http://proactive.activeeon.com (2011)
18. Oracle: Future Interface Documentation. http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html (2012)