Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 281 (2011) 127–141

www.elsevier.com/locate/entcs

Detecting and Coordinating Complex Patterns of Distributed Events with KETAL

Luis Daniel Benavides Navarro, Andrés Barrera
Kiyoshige Garcés, Hugo Arboleda

Departamento de Tecnologías de Información y Comunicaciones
I2T Group, Universidad Icesi
Calle 18 No. 122-135 Pance, Cali, Colombia
daniel@rainconcept.com {andres.barrera| oscar.garces| hfarboleda}@icesi.edu.co

Abstract

This paper presents an event-based kernel library designed to explicitly construct and coordinate complex interactions and communication patterns in distributed applications. The library integrates facilities for explicitly defining complex event patterns, detecting events in distributed systems, and validating sequences of events having into account causal ordering. Concretely we present the following contributions: *i)* An analysis of non trivial scenarios found in distributed applications in order to formulate a set of requirements and restrictions for a kernel event-based library, *ii)* the design and implementation of the library supporting the detection and coordination of complex event patterns and the support of causal manipulation of distributed events, *iii)* a qualitative evaluation of our approach showing how this library can be used to build a sophisticated distributed aspect oriented language.

Keywords: Distributed event model, event patterns, causality, automata.

1 Introduction

A distributed system consists of a collection of autonomous computers, so called nodes or hosts, connected through a network and distribution middleware, which enables nodes to coordinate their activities and to share the resources of the system, so that users perceive the system as a single and integrated computing facility [23].

The implementation of distributed systems is a difficult task, in part because, current mainstream approaches and tools do not provide mechanisms to explicitly define, coordinate and implement distributed algorithms and communication patterns. Instead, programmers are forced to design defensibly and to create isolated, complex and disjoint software components to implicitly control the behavior and communication protocols of distributed applications. Consider, for example, a 3 tier application. On such an application, communications are restricted to contiguous layers and, in most cases, such a restriction implies that communication between two

doi:10.1016/j.entcs.2011.11.030

layers is done through a unique virtual entity (*i.e.*, simulating only one computer). This kind of defensive design restrictions simplifies greatly the implementation of distributed applications. However, it still requires distributed algorithms and communication patterns to be encoded between at least two components: one in the layer starting the communication and the other in the layer receiving the communication.

To address the limitations of current tools and approaches for distribution, several design techniques and architectural styles have been proposed, see for example design patterns [9,1] and integration patterns over messaging middlewares [10]. However, most of these approaches are catalogs of best practices, idioms, and recommended usage scenarios that deal with the restrictions of current tools an do not improve on the tools main abstractions. Furthermore, recent research has shown that the usage of these best practices and patterns do not improve on the actual complexity of resulting code. For example, Benavides et al. showed in [6] that the implementation of complex communication patterns used to replicate dynamic session information on clusters of JEE application servers, resulted in tangled and scattered code difficult to understand and maintain. Even tough, the communication patterns were clearly defined, well documented, and developed according to current best practices.

Recent research has proposed the development of new tools and conceptual frameworks to support expressive and sophisticated constructs to address distributed programming [12,7,11,20,6]. Most of these tools propose an event model and several constructs to match and manipulate streams of distributed events. These tools need solid building blocks to allow the correct, reliable, and efficient implementation of compilers, tools, and runtime frameworks. This paper presents a kernel event-based library designed to explicitly construct and coordinate complex interactions and communication patterns in distributed applications. Concretely we present the following contributions:

- An analysis of non trivial scenarios found in distributed applications in order to formulate a set of requirements and restrictions for an event-based kernel library.
- The design and implementation of the library addressing the detection and coordination of complex event patterns with support for causal manipulation of distributed events.
- Qualitative evaluation of our approach showing how this library can be used to build a sophisticated distributed aspect oriented language [1].

This document is structured as follows. First, in Section 2 we motivate our work by means of detailed analysis of non trivial scenarios found on distributed applications. Then, we present the design considerations for the kernel library in section 3. Section 4, discusses the implementation of the kernel library to support the proposed constructs. Section 5 presents a qualitative evaluation by means of the implementation of a Distributed Aspect Oriented Language. Finally, section 6

[1] This project is part of the AWED [5] language reimplementation.

reviews related work and section 7 concludes.

2 Motivation

In this paper, we argue for a mechanism to explicitly construct and coordinate complex interactions and communication patterns on distributed applications. In this section, we present three non-trivial scenarios that show what kind of problems arise when dealing with distributed events on distributed applications. First, we present an abstract scenario where a set of events are communicated between the nodes participating in the application. Then, we discuss two concrete cases in the context of typical debugging and testing tasks of distributed middleware.

2.1 Looking for event patterns

Let's suppose we have three computers, `Node A`, `Node B` and `Node C`, on which we deploy a distributed system to manage a replicated stack with basic operations implemented. The stack provides `push` and `pop` services and additionally it provides and starting and stopping protocol to control when the stack is replicating (`on` and `off` services). The `Node A` is the responsible of executing the `on` and `off` operations, the `Node B` is the responsible of executing the `push` operation and the `Node C` is the responsible of executing the `pop` operation. According to the defined integration protocol, the behavior of the distributed system must be the following: each time an event occurs in a node, a coordination message is passed to the others in order to inform about the event that just happened and coordinate the replicated stack. Figure 1 presents such behavior. The first event, `on`, occurs in the `Node A` so that the stack is started. Two messages are passed informing to `Node B` and `Node C` that the stack is turned on. After the `Node B` and `Node C` detect the messages, the second event, `push`, occurs in the `Node B` so that the stack has its first inserted element. Two messages are passed informing to `Node A` and `Node C` that an element was added to the stack. After the `Node A` and `Node C` detect the messages, the third event, `pop`, occurs in the `Node C` so that the stack is now empty again. Two messages are passed informing to `Node A` and `Node B` that an element was removed from the stack. Finally, and after the `Node A` and `Node B` detect the messages, the fourth event, `off`, occurs in the `Node A` so that the stack is turned off. Two messages are passed informing to `Node B` and `Node C` that the stack is not available any more. The `Node B` and `Node C` detect the messages.

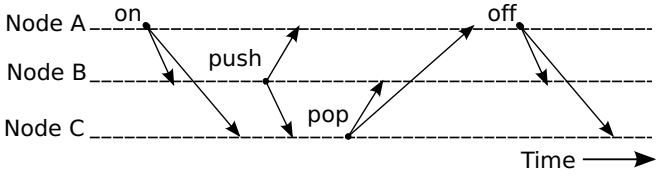

Fig. 1. Example of distributed stack over three nodes.

This would be a trivial scenario if the available system events are not constrained by the current state of the stack. This, however, is not the case. Available operations

in stacks are restricted by previous executed operations. For instance, the stack only accepts a `push` if the `on` was already executed. Then, every node, after receiving a request, must validate the current state of the local stack before changing it. This implies to check the relationships between distributed events.

By using the current techniques, intricate relationships between distributed events have to be defined in terms of conditions on the execution state of individual hosts. Thus, relationships involving multiple hosts have to be expressed using complex and isolated encodings that are difficult to understand and to maintain.

2.1.1 Ordering problems on event patterns: false positives/negatives

In the above scenario we suppose that no matter what operation is requested in a node, the event is triggered and every remaining node is immediately notified with the purpose of coordinating the distributed system. This behavior supposes determinism in the arrival order of coordination messages; first coordination message sent, first coordination message received. The determinism that presents the scenario is not the common case in distributed systems, which have to deal with concurrent processes, random delays in data transfer or denegation of network services, between other usual cases. Figure 2 presents an example where the problem of nondeterminism in the detection of event patterns is illustrated. In the figure the first event, `on`, occurs in the `Node A` and two integration messages are passed in order to notify, to `Node B` and `Node C`, that the stack is turned on. The `Node B` detects the message. However, the `Node C` detects the message after it detects a message coming from the `Node B` to inform that a `push` occurred. Due to this interleaving of events, the event pattern detected by the `Node C` is <`push`,`on`>, that is different from that detected by node `B`, <`on`,`push`>.

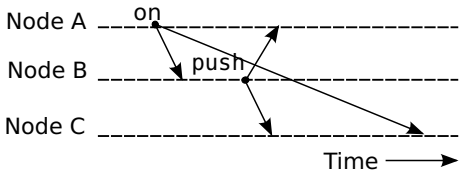

Fig. 2. Example of nondeterminism in event's ordering.

In distributed systems, event patterns frequently are of interest only if they occur as part of specific and well defined execution traces but not in the presence of different interleavings of the events that are part of those traces and occur due to non-deterministic executions. For instance, in our example, the `Node C` could be interested in the event pattern <`push`,`push`,`push`> with the purpose of consuming the items on the stack: <`pop`,`pop`,`pop`>. Besides, the `Node A` could be interested in the event pattern <`push`,`push`,`push`,`pop`,`pop`,`pop`> with the purpose of turning the stack off: <`off`>. In the presence of interleaving of events, it could happen that some event patterns are never recognized or they are recognized erroneously. We refer to the problems as:

- Detection of *false positives*: wrong sequence of events are detected and mapped to predefined event patterns.

- Detection of *false negatives*: correct sequence of events, mapping to predefined event patterns, are not detected.

Figure 3 illustrates the two situations. Let's suppose the `Node C` is interested in the event pattern with the sequence <`push`,`on`>. This sequence never occurs; however, the `Node C` detects the sequence, which is a false positive. Now, let's suppose the `Node C` is interested in the event pattern <`push`,`pop`>. This sequence does occur; however, the `Node C` does not detect the sequence because it detects <`push`,`on`,`pop`>, this missing pattern is what we called a false negative.

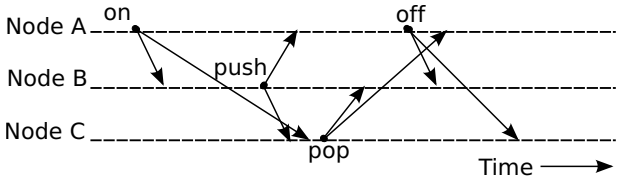

Fig. 3. Illustration of the false positives/negatives problems.

Currently, several approaches to define and validate relationships among constraints on events in distributed systems have been proposed. For instance, data path expressions for concurrent programs [21], causal event relationships based on logical clocks [2,8,15], and control-flow based event relationships [16]. However, the declarative flavor of these proposals have not been integrated into mainstream middlewares. Hence, besides that relationships involving multiple hosts have to be expressed using complex and isolated encodings, independent components have to be aware of catching false positives/negatives and deal with them.

2.2 Concrete example: Testing and Debugging distributed middleware

When fixing an error in the context of a software application, at least the following two actions are executed: first a developer creates a debugging session to reproduce and fix the problem; then, the developer writes a test case in order to reproduce the error, on each regression test run, and as such grant it won't appear again. In [20] it was shown that JBoss Cache [13] suffered from deadlock errors in the transaction algorithm (Two phase commit protocol), in presence of multiple threads accessing the same cache position. These errors are very difficult to fix because the the two actions described above are not easily implemented.

With respect to the debugging session, for example, the error was difficult to reproduce because the incorrect interleaving of events, that generated the deadlock, were reproduced randomly by the scheduler. Thus, in a common debugging session with limited number of threads and big delays imposed by the debugger the error is almost impossible to generate. Thereby, since current debugging tools do not provide abstractions to concisely express such cases, programmers once again have to manually encode integration information in order to deduct the distributed state. This tactic implies applying programming tricks, such as polluting the original code with state information, in order to track the necessary control flow dependencies. Often these debugging tasks can be much facilitated by ensuring that occurrences of

events obey strict ordering constraints. However, current debuggers do not support such facilities and programmers have to resort to encodings of distributed state and coordination information.

The situation is similar when writing a test case for such error. A common idiom used when testing errors involving random interleaving of events is to trigger several threads at the same time, in order to reproduce the error randomly. For example, in JBoss Cache there is a test case with such an idiom. Concretely, the test case uses two caches, actions on the first cache are replicated into the second cache by means of the replication framework. The test case triggers multiple workers in multiple threads. Each worker starts a transaction, puts a value in the cache and commits the transaction. The deadlock occurs when a worker, after a successful prepare phase of the two phase commit protocol, commits a transaction and releases the lock over the source cache after the local commit, but before completing the final commit phase with the remote caches. There, other workers may interleave their transaction operations, in particular, acquire the lock at the same cache position and thus preclude the first transaction to terminate its remote commit phase, thus entering a deadlock situation, because no worker can acquire all necessary local and remote locks anymore.

In order to reproduce the deadlock, the test case has to be run a number of times. However, it is possible that after several tests the deadlock is not reproduced. This is because current techniques does not provide constructors to program and coordinate cohesively the communication behavior, such as sequences of intermediate synchronous or asynchronous calls. Then, the construction of effective test cases that ensure that some errors are systematically reproduced is a tedious and difficult task. Having control over ordering of events may improve greatly the creation of deterministic test cases.

3 Design Considerations

This section discusses the main considerations and assumptions we did in order to design the kernel library. We first discuss the event model, then we discuss how patterns of events may be detected using a model based on finite state automata. Finally, we consider a dynamic model for time constrains and several usage scenarios for the library.

3.1 The event model: Detecting event based patterns

The first design consideration we discuss is the required event model. Based on the motivation above, we need a model for distributed events. Such a model have to do a precise distinction between messages and events. Events are atomic actions occurring on a specific process at an individual node. Messages, instead, are the packets of information sent from a specific process, on an individual node, to other process running on another node. Examples of possible actions are a method call, sending a message, and receiving a message. Note that the occurrence of an event can only be notified to other nodes by means of messages sent through the network.

It has to be considered also that events are originated in a context. Such a context may refer only to static information, *e.g.*, the value of variables and parameters at the moment that the event occurred, but it could also refer to more complex dynamic information like the control flow history (*i.e.*, trace) of an application. In our model, it is required the context to include information about physical localization, it means, the node where an event occurs. Thus, when information about an event is sent through the network, the information regarding localization must be attached.

Finally, the model has to expose a fully distributed architecture, *i.e.* without centralized processors or bottlenecks, and with all events being notified to the other nodes. In this model each node participating in the application may receive and send messages regarding the occurrence of events. The model is also dynamic, meaning that nodes may enter and leave the distributed application at any time. As such, there is not any centralized server or broker managing the notifications. For the purpose of designing the kernel library we consider that all events occurring on any node are notified to the other nodes. Note that this last restriction is just there to help us with design, leveraging the burden of what events should be distributed. In our case we consider that all events are distributed. Afterwards, the reader will see that this restriction is not made explicit for the library to work, and that in concrete applications (see aspect language implementation on section 5) intelligent selection of events makes the quantity of distributed events very low.

With the event model in place we can start talking of event patterns. The simple case of a pattern of events is the atomic event, such an event may now be detected on any node, and information from its context, including localization, may be extracted. This case can be extended to detect sequence of events. Thus, node may listen, *e.g.*, for the sequence of events <`callMethod A on host1`,`callMethod B on host2` >(detect a call to method `A` on hots number 1 and then detect a call to method `B` on hots number 2). However, these patterns are simple and we are interested in more complex patterns, next section introduces a model to handle more complex patterns of events.

3.2 The pattern model

In the previous section, we introduced the design considerations for a model where an atomic event can be detected on any node participating in a distributed application. Now, in order to detect a complex pattern, we propose to use a finite state automata-based model. This model will consider that on each node, interested in a particular pattern of events, an automaton will be deployed. The automaton will consume event notifications to trigger the transitions.

The example on figure 4 shows a replication protocol for a replicated stack, similar to the one motivated in Section 2. In the example, a simple stack and a monitoring automaton is deployed on each node. The automaton is in charge of replicating events occurred on remote nodes. The automaton in the figure will detect events `start` and `stop` on the local machine (the local machine is a relative designation predicating over the deployment machine of each automaton), these

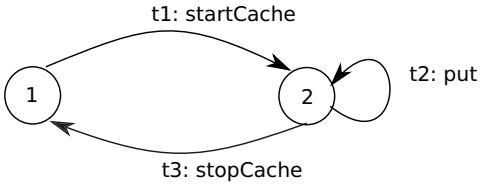

Fig. 4. Simple replication protocol for a replicated stack

events will trigger transitions `t1` and `t3` respectively, starting and stopping replication. Transition `t2` will be triggered by remote events `push` or `pop`. Note that we have not bound explicitly a behavior to transitions, however we can do it, for example by binding the necessary code to replicate on the local stack the detected `push` and `pop` remote events.

The example above shows the main assumptions we do on the model in order to deal with event patterns:

- Automata instances are deployed on each node participating in the application.
- Automata will consume events to trigger transitions. Those events may be remote events.
- An action may be bound to specific transitions.
- Actions bound to transitions are executed on the local machine.
- In order to consume an event an automaton may predicate over the event localization (where it occurred). Those predicates may refer to relative localizations like `localhost` or `not localhost`

Until now we have not consider time and ordering issues in the model. The following section addresses such considerations explicitly.

3.3 The Dynamic Model: Ordering of distributed events

We have presented the design considerations of a model where an atomic event, or patterns of events, can be detected on any node participating in a distributed application. The model relies on the implicit assumption that the events are consumed in the same order that they are produced. However, as we mention before, event notification is done trough messages sent over the network, and the network is not a reliable resource: several delays and not determinism is introduced. Furthermore, to grant performance and reliability the order of messages is not granted in most protocols. Several, algorithms and solutions have been proposed to solve this issue. A first approach is to have synchronized clocks on each node and force ordering of messages. This approach, however, has been shown to be very costly and complex to implement and maintain. Other approaches relying on partial orders, *e.g.* logical clocks [15] and vector clocks [18], propose a more lightweight solution where some specific ordering situations cannot be determined. But, currently, the most popular solution is that distributed applications are defensively designed so they do not assume message ordering in their algorithms.

Hence, the library must support the model as stated in previous sections, but

it has to support ordering algorithms. This algorithms will be very useful when dealing with complex event patterns; *e.g.*, those involved in distributed test cases and complex break points as studied by [20] and presented in Section 2. For this research we will support causal ordering of events, as stated in the motivation section. Such algorithms will be based on Mattern's vector clocks [18]

According to these considerations, the model should be extended as follows:

- Transitions on automata should support guards. Guards are boolean conditions that are evaluated before performing a transition on an automaton.
- The library should support *causal* and *concurrent* guards. Thus, events will be consumed only if they respect these conditions with respect to the partial ordering defined by causality relation (causality relations' partial ordering are implemented using vector clocks).
- Finally, the model must support special constructs to grant event ordering. Thus, causal relation will not only be evaluated but it will also be granted, by means of message reordering before automata consumption.

3.4 Concrete kernel usage scenarios

We now discuss four concrete usage scenarios for the kernel library. Such scenarios are created form the point of view of a programmer. *The first scenario* regards automata creation by means of a regular expression. Thus, a programmer could use for example the expression `aabc+adc` in order to define an automaton. *The second scenario* refers to consumption of full words of events. According to the design considerations we have introduced until this point, our automaton consumes events and not characters. In order to accomplish the *the first scenario*, we must include a new design consideration that includes mapping of events to characters. Hence, to consume events, the library must accept a word of events passed inside an ordered data structure, *e.g.* a vector, and mapping it to an ordered data structure of characters. *The third scenario* concerns to the validation of event patterns against automata. Once a word is received the library must answer if the word is matched by an automaton. Finally, *the fourth scenario* is related to explicit creation of automata. The library must allow programmers to create an automaton by providing an event alphabet, an initial state, a set of states, a set of transitions, and a set of final states. The consumption of events in this case may be done atomically, one event at the time. The automaton, in this scenario, may answer the current state of the automaton.

4 Implementation

In this section we describe the main components of our event-based kernel library. We first present the architecture, its elements, and how such elements implement the model described in the previous section. We then, show in detail how we build the automata framework to support the detection and management of patterns of distributed events. The library code can be found at [17].

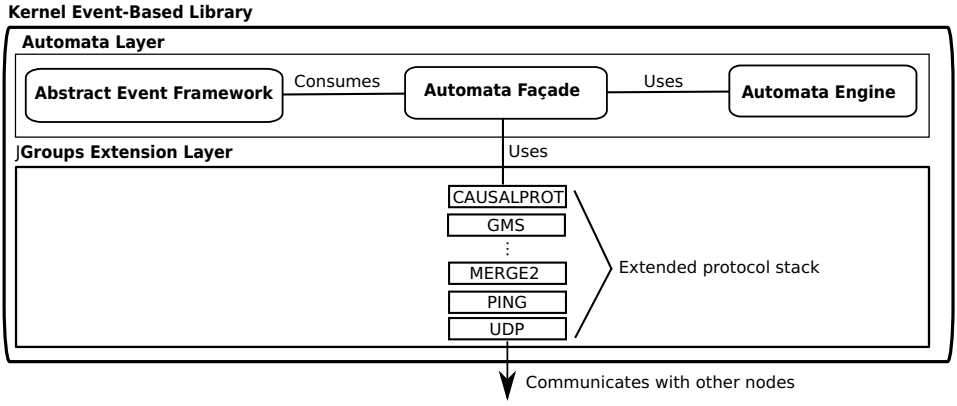

Fig. 5. Architecture of the kernel event-based library

4.1 Architecture

Figure 5 presents the four main components of the kernel library architecture: the `Abstract Event Framework`, the `Automata Facade`, the `Automata Engine`, and the `Distribution Layer`. The `Abstract Event Framework` is basically a set of interfaces that allow the developer (user of the library) to adapt existing code in order to be modeled as events that are consumable by the automata defined by the library. The `Automata Facade` provides the abstractions to manipulate directly the definition and execution of an automaton (see next section for details). The `Automata Engine` is in charge of processing actions over the automata according to automata theory. Currently, we are using as automata engine the automata library provide by Anders Møller et al. [19]. Note that this component may be changed for other one without affecting the interface and services provided by the library.

The `Distribution Layer` provides the main abstractions to distribute event messages and listen to event messages sent by other nodes. The layer provides a distributed architecture (*i.e.* with no centralized component) based on group communication (see JGroups [14]). Each node on the distributed application must have a deployed instance of the kernel library.

In order to handle the causality predicates and causal order of messages, we have developed two protocols that are configurable on the JGroups protocol stack (see Figure 5, `JGroups Extension` component). By default, JGroups provides several protocol implementations including `GMS`, `MERGE2`, `PING`, and `UDP`. The protocols developed are depicted on the figure as the box labeled `CAUSALPROT`. The first, protocol adds causal information to event messages, so the receiver node can determine if messages arrive in causal order of if they are concurrent. This protocol avoids false positives; *e.g.*, if two messages are inverted, the protocol will detect that the second message is not in the right order and it will not be consumed. The second protocol adds also causal information to event messages, but, before delivering a message to the `Automata Layer` it orders messages according to the causal partial order. Thus, this protocol forbids false positives and avoids false negatives.

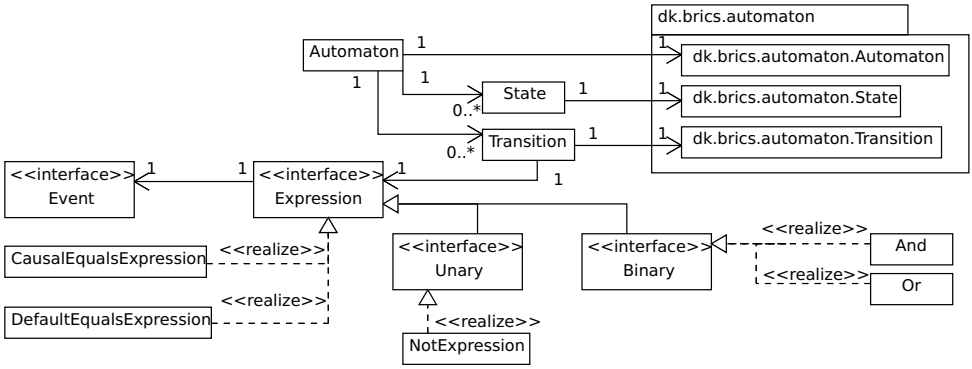

Fig. 6. Class diagram of the `Automata Facade` component and its relation with the classes on the `Automata Engine` component.

4.2 The Automata Layer

Figure 6 shows the main classes we implemented in the `Automata Facade` and `the Automata Engine` components. The classes `Automaton`, `State`, and `Transition` represent the main abstraction of an automaton in the `Automata Facade` component. The `Facade` provides a set of services that allow programmers to use an automata to consume distributed events. The classes in the `dk.brics.Automaton` package are the classes from the Automata Engine (see Møller et al. automata library [19]). Transitions may be attached to an expression over events (see interfaces `Expression` and `Event`). When the transition is attached to an expression, the user of the library decides how events are consumed and when the transition is made. Even tough, in principle, events cannot be attached to transitions directly, the library provides methods to mimic such relation. When events are attached to transitions, what happens is that a `DefaultEqualsExpression` is created under the hood. Such expression compares a arriving event with the defining event, an this is a behavior that mimics consumption of events as transitions.

Having the possibility of adding expressions to transitions provides immediate support for guards. In order to build a sophisticated guard expressions the programmer may use `Not`, `And`, `Or` constructs. For this, the library provides a hierarchy of expressions constructs including the unary `Not` expression and the binary expressions `And` and `or`. However, for flexibility, the user may create its own expressions library according to its needs.

5 Evaluation: Designing a Distributed aspect language with complex pointcuts

We have used our Kernel Library to build support for complex pointcuts on a sophisticated Aspect Oriented Language with explicit support for distribution (AWED [5,20]). In this section we make a brief description of AWED main concepts, and make a qualitative evaluation showing how the library is used to implement those concepts.

AWED proposes three main concepts: *Aspect*, *Distributed Pointcut*, and *Distributed Advice*. Aspects are class like constructs that are used to define crosscutting concerns. Those aspects, contain pointcut definitions and advices. Distributed Pointcuts are declarative constructs that are used to match specific events over the execution trace of a distributed program. AWED proposes a pointcut language that is sensible to event localization. For instance, AWED not only provides pointcuts to match events like method calls, methods execution, variable update, but it also provides pointcuts to match events that occurred on an specific host or group of hosts. Advices are method like constructs that are bound to pointcuts and that are executed once a specific pointcut matches an event.

The set of pointcuts proposed by AWED includes constructs for sequence of events (similar to our automata) and a sophisticated synchronization model. The synchronization model includes a synchronous matching semantics, where the base program (the one generating events) is paused, once an event is matched and until the advice finishes its execution. The language also provides an asynchronous semantics, where advices and base programs are executed concurrently. Both semantics may be executed in a distributed fashion. Thus, for example, the base program may be paused in one machine while the corresponding advice is executed in other machine. The model is completed by a set of constructs that allows programmers to build pointcuts predicates including causal relations and concurrent relations.

To implement the constructs described above we have first modified the pointcut implementation in order to use the `Event Framework`. Then we have modified AWED's compiler and `Sequence` pointcut to build and use automata defined by the `Automata Facade`. To address causal predicates we have mapped the `Causal` and `Conc` (Concurrent) modifiers into expressions using the constructs provided by the `Automata Layer`. The most complicated requirement to implement is the synchronization model. In order to implement such model, we have created a mechanism for intelligent weaving. Such process first look for pointcut definitions and identifies specific points at the code that may generate detectable events. At those points the weaver adds the glue code to broadcast events and, depending of the selected synchronization mode, pause or execute concurrently the base program. When executing the base program concurrently the model was extended to provide future like semantics for synchronization.

6 Related Work

In order to select an `Automata Engine` component, we studied several libraries and applications which target is the modeling of situations using automata and regular expressions. In this section, as a representative piece, we present four of them. We also discuss current distributed event-based models.

Anders Møller's automata library [19] was developed at Aarhus University. The library allows developers to define a finite-state automaton with regular expression operations. There exist two ways to create an automaton in such a library: *i)*

by using a regular expression that describes the language and, *ii)* by defining it explicitly; it is, we must create states and the set of transitions between them. The library provides, between other facilities, functionality to determine whether a given word or string is a member of some particular language, an automaton is deterministic or whether the recognized language is finite. JFlap [22] is a package that provides facilities to create deterministic or non-deterministic automata. It also includes functionality to manage stacks, moore machines, turing machines and regular expressions. Its most remarkable features are its user interface and its facilities for formal languages and automata theory learning. Jakarta RegExp [24] is a library devoted to create and manipulate regular expressions in order to define domain specific languages. The library does not include the concepts of state and transition; hence, the only way to determine if a received string is a member of a language is by consuming characters one by one, and recursively to establish the string membership in the context of the language. JAuto [4] is a library providing facilities for manipulation of finite-state automata within the Java platform. Its API includes functionality for creating automata, converting them to/from an external source, testing of various relations between automata, and evaluating unary and binary predicates over automata. In [3], Bailly et al. presents a model of distributed components that includes an event model on the component traces. This work has used JAuto as a means for components composition in order to preserve behavioral contracts. As mentioned above, these four libraries are representative examples of tools implementing automata and regular expressions theory. Nevertheless, none of them are devoted neither to recognize event patterns nor to deal with distributed events. Besides, RegExp, by failing to include the concept of state, is not able to control the transitions in the consuming of events, which is part of the design considerations we argued in Section 3.

Regarding Composite Event Detection several approaches have been proposed, see for example [7,12,11]. Those approaches provide a means to create predicates over a set of events. In principle, these predicates define a sub set of the set defined by our pattern model. For example, they may detect sequence of events but they do not have full support for regular expressions over event traces. Another point of comparison with our proposal is the synchronization model. Most event based approaches use asynchronous interaction, in contrast our model does not impose such restriction and, as shown in the evaluation section, allows the library to be used also in applications with synchronous interaction models.

7 Conclusions

In this paper, we argued for a mechanism to explicitly construct and coordinate complex interactions and communication patterns in distributed applications. We have presented an analysis of non trivial scenarios found in distributed applications and we formulated a set of requirements and restrictions for a kernel event-based library, we designed and implemented the kernel library, which supports detection and coordination of complex event patterns having into account causal manipula-

tion of distributed events, and we evaluated our approach. We included in the kernel library constructors to encapsulate inside well defined components the coordination of distributed applications, instead of delegating the coordination to disjoint components; for this, the architecture of the library is based on an automata library implementing basic automata operations. Our evaluation has shown that our library can be used to build or improve a sophisticated distributed aspect oriented language such as AWED.

Our work prepares the way for several leads of future work in two main areas, model and appliction. For model extension we will explore more complex patterns and concepts, e.g. epsilon transitions, push down automata. With respect to applications, we will study the implementation of a fully distributed debugger for Java, and the development of dynamic graphical modeling tools to generate software artifacts.

References

[1] Alur, D., D. Malks and J. Crupi, "Core J2EE Patterns: Best Practices and Design Strategies (2nd Edition)," Prentice Hall, 2003, 2 edition.

[2] Anderson, J. H., *Lamport on mutual exclusion: 27 years of planting seeds*, in: *PODC'01: Proc. of the 20th annual ACM symposium on Principles of distributed computing* (2001), pp. 3–12.

[3] Bailly, A., M. Clerbout and I. Simplot-Ryl, *Component composition preserving behavioural contracts based on communication traces*, in: *Implementation and Application of Automata*, LNCS **3845**, Springer Berlin/Heidelberg, 2006 pp. 54–65.

[4] Bailly, Arnaud and Roos, Yves , *JAutomata home page*, latest visit on may 2011 (2011).
URL http://jautomata.sourceforge.net/

[5] Benavides Navarro, L. D. and M. Shüdholt, *AWED home page*, latest visit on may 2011.
URL http://awed.gforge.inria.fr/

[6] Benavides Navarro, L. D., M. Südholt, R. Douence and J.-M. Menaud, *Invasive patterns for distributed programs*, in: *Proc. of the 9th International Symposium on Distributed Objects, Middleware, and Applications (DOA'07)*, LNCS (2007).

[7] Eugster, P. and K. Jayaram, *Eventjava: An extension of java for event correlation*, in: *ECOOP'09 – Object-Oriented Programming*, LNCS **5653** (2009), pp. 570–594.

[8] Fowler, J. and W. Zwaenepoel, *Causal distributed breakpoints*, in: *Proceedings of the 10th International Conference on Distributed Computing Systems (ICDCS)* (1990), pp. 134–141.

[9] Gamma, E., R. Helm, R. Johnson and J. Vlissides, "Design patterns: elements of reusable object-oriented software," Addison-Wesley Professional, 1995.

[10] Hohpe, G. and B. Woolf, "Enterprise Integration Patterns: Designing, building, and deploying messaging solutions," Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.

[11] Itzstein, G. S. V. and D. A. Kearney, *The expression of common concurrency patterns in join java*, in: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, Volume 2*, Las Vegas, Nevada, USA, 2004, pp. 1021–.

[12] Jayaram, K. and P. Eugster, *Scalable efficient composite event detection*, in: D. Clarke and G. Agha, editors, *12th International Conference on Coordination Models and Languages (COORDINATION 2010)*, LNCS **6116**, Amsterdam, The Netherlands, 2010, pp. 168–182.

[13] *JBoss Cache home page*, latest visit on may 2011 (2011).
URL http://labs.jboss.com/jbosscache

[14] *Jgroups home page*, latest visit on may 2011 (2011).
URL http://www.jgroups.org/

[15] Lamport, L., *Time, clocks, and the ordering of events in a distributed system*, Commun. ACM **21** (1978), pp. 558–565.

[16] Li, J., *Monitoring and characterization of component-based systems with global causality capture*, in: *23th Int. Conf. on Distributed Computing Systems* (2003).

[17] Luis Daniel Benavides Navarro, Andrés Barrera, Kiyoshige Garcés, Hugo Arboleda, *Ketal–kernel event-based automata library home page*, latest visit on may 2011 (2011).
URL http://www.icesi.edu.co/driso/ketal.php

[18] Mattern, F., *Virtual time and global states of distributed systems*, in: *Proceedings of the international Workshop on Parallel and distributed Algorithms*, Chateau de Bonas, France, 1988.

[19] Møller, A., *dk.brics.automaton – finite-state automata and regular expressions for Java*, latest visit on may 2011 (2010).
URL http://www.brics.dk/automaton/

[20] Navarro, L. D. B., R. Douence and M. Südholt, *Debugging and testing middleware with aspect-based control-flow and causal patterns*, in: *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08 (2008), pp. 183–202.

[21] Ponamgi, M. K., W. Hseush and G. E. Kaiser, *Debugging multithreaded programs with MPD*, IEEE Software **6** (1991), pp. 37–43.

[22] Rodger, S., *JFLAP home page*, latest visit on may 2011 (2011).
URL http://www.cs.duke.edu/csed/jflap/

[23] Tanenbaum, A. S. and M. v. Steen, "Distributed Systems: Principles and Paradigms (2nd Edition)," Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[24] The Apache Software Foundation, *Jakarta RegExp home page*, latest visit on may 2011 (2011).
URL http://jakarta.apache.org/regexp/index.html