An Event Based programming language for Runtime Monitoring and Dynamic Instrumentation of Concurrent and Distributed Programs

Luis Daniel Benavides[1], David Durán[2], Camilo Pimienta[2], and Hugo Arboleda[2]

[1] Escuela Colombiana de Ingeniería Julio Garavito
luis.benavides@escuelaing.edu.co
[2] Universidad Icesi, i2T Research Group, Cali, Colombia
{dduran,cfpimienta,hfarboleda}@icesi.edu.co

Abstract. In this paper we introduce EKETAL, an event based programming language for runtime monitoring and dynamic instrumentation of distributed and concurrent applications. We argue that development, maintenance and evolution of distributed applications can be greatly improved by such a language. To support this claim, we first present the programming model and corresponding compiler implementation (compiler generates distributed AspectJ's code), then we present micro-benchmarks of the runtime infrastructure and a qualitative study of the usage of the language for debugging and testing *liveness* and *datarace* problems found in BigData middleware.

Keywords: Debugging, testing, event patterns, runtime monitoring, event based languages, distributed applications, liveness errors, datarace error.

1 Introduction

The advent of new technologies such as multi-core processors, virtualization, platform as a service clouds, and mobile devices, has changed the requirements of distributed applications. Multi-tenancy, resiliency, responsiveness, scalability, and elasticity, are quality properties that must hold on applications deployed over thousands of servers and millions of devices with disparate middleware stacks. The development, maintenance and evolution of such applications has forced the resurgence of simpler programming models for distribution and concurrency. Patterns for asynchronous remote calls, message driven communications, and event oriented architectures have emerged as programming paradigms for such heterogeneous and complex environment.

Consider for example the development of highly scalable and transactional services with non-blocking operations, in order to attend several millions of users (e.g., Twitter or Whatsapp). In those cases, it has been documented the usage of technologies such as Scala and the Java Virtual Machine (JVM), to attend Twitter back-end [8], and the usage of Erlang to implement the WhatsApp messaging

infrastructure [11]. These two languages, Scala and erlang, provide constructors for asynchronous messaging, futures, functional programming, actors and other abstractions related to patterns of asynchronous messaging and event oriented architectures. Even tough, the examples mentioned above support simpler programming models, there is still a conceptual mismatch between the abstractions used to implement the distributed applications, and the applications used to support their development and maintenance.

Contributions. In this paper we present an event based programing model with distributed and concurrent abstractions that facilitates runtime monitoring of heterogeneous distributed applications. Concretely, we present three contributions. First, we introduce EKETAL, an event based programming model and language for runtime monitoring and dynamic instrumentation of programs adopting ideas proposed in [14, 2]. The model use actor-like constructs to monitor and modify the execution of distributed and concurrent programs. It uses Deterministic Finite automaton constructs combined with a predicate language to support detection of complex event sequences, it supports causality, ordering of messages as well as futures and other mechanisms for synchronization. Second, we present the implementation of a compiler to support such programming model. The compiler translates the EKETAL code into AspectJ code with support for distribution using groups communication. Finally, we show performance and usage evaluation of these techniques in order to test and debug actual *liveness* and *datarace* problems found in industrial middleware and big data frameworks.

The paper is organized as follows. First, in Section 2, we present the state of the art. Second, in Section 3, we introduce an event model and define an event based programming language for runtime monitoring and instrumentation. Third, In Section 4, we present the implementation of of a compiler for the proposed language. In Section 5, we validate our proposal studying concrete examples of *dataraces* and *liveness* errors in industrial level concurrent and distributed applications (e.g., Hadoop and *JBossCache*[21]). In section 6, we present the conclusions and discuss future work.

2 State-of-the-art

In this section we first discuss the state of the art of event based frameworks and languages by means of a taxonomy of those approaches. Then, we present work related to run-time monitoring and runtime verification, particularly those using aspect oriented techniques.

2.1 Event oriented languages and frameworks

To present the state-of-the-art we start from the classification of event-oriented programming languages proposed in [12]. This classification considers three characteristics: the amount of allowed repetitions of an event, the amount of events

that a pattern can correlate, and the amount of allowed events in predicates. Using this categorization, at the base we found the *observer* pattern with one event repetition, one event in the pattern, and zero event predicates. At the top of the classification, we found languages such as *EventJava* [9] and database systems such as *Cayuga* [7] with multiple event repetitions, multiple correlated events, and multiple events in predicates. Although these systems can detect complex events, they do not directly support the synchronization and causal ordering of events. We will show that our approach provides a richer predicate language with explicit constructs to support concepts from distributed and concurrent programming.

A different set of proposals has used formal methods and formal calculi to support event models. Languages derived from the join calculus [10], such as Polyphonic C#, which [3] support one repetition per event, several events in the pattern, and no predicate. They support asynchronous and synchronous calls to reactions (chords in Polyphonic C#), and a fair finite state machine can be encoded. Unlike *EKETAL*, neither of these approaches offer explicit support for distribution and pattern matching in a distributed setting, nor support causal relations between events. EVENT-B [1] proposes a full methodology to develop software based on formal specification and a formal language based on set theory. This language offers a very general framework for modeling event-based software and has been validated by implementations of the language [4] and of code generators for mainstream languages such as Java [19]. The EVENT-B proposed methodology provides safety properties for concurrency problems similar to those discussed here. However, it lacks explicit constructs to model localization and does not support the explicit ordering of messages.

Other types of frameworks such as *Thread-Sanitizer* [20] allow the detection of *data races* in non-distributed concurrent systems and are of great help to professional programmers. However, they do not allow the definition of sequences by a user but instead focus on common problems that cause *dataraces* in only one computer.

2.2 Runtime Monitoring and Aspect Oriented Programming

We have also investigated the applicability of concepts found in aspect oriented languages to create runtime monitoring frameworks. *AWED* [2] is a language with explicit support for automata, guards in the transitions of these automata, predicates for the detection of complex patterns over traces of execution and synchronization mechanisms. Moreover, *AWED* addresses the debugging and testing of deadlock errors. The research presented here extends this study by analyzing a more complete set of concurrent and distributed errors (liveness and data race errors) and proposing a more general event model supported by an event-based language. Other researchers have explored the applicability of similar techniques to create runtime verification tools, e.g., see [6].

The concept of Monitoring Oriented Programming (MOP) was introduced in [5] in order to generalize and formalize a set of runtime monitoring tools that used a formal language to express properties, that a base program must hold,

and reactions when those properties where validated or invalidated. This work has been extended to support multithreaded programs [16]. Our work follows this general model, extending further these ideas, incorporating event oriented concepts as programming paradigm, and providing explicit constructs to reason and predicate about distributed applications. Other works have studied general models for implementation and expressiveness of this kind of frameworks. In [22], authors studied ways to implement method slots, a construct that encompasses the implementation of methods (from the object-oriented paradigm), events, and advices (a method like construct from the aspect-oriented paradigm). In [15], the authors address language expressiveness power and discuss ways to achieve Turing completeness in different components of the language. They divide the language into three essential components, i.e., pattern language, matching process, and advice mechanism (reaction mechanism), and study how to improve expressiveness. These are more general models than EKETAL, but the actual implementation of such models using abstractions for distribution and concurrency remains a matter of future work.

3 A general event based language for detection and manipulation of Complex Event Patterns

3.1 Event model

The language was designed to monitor and modify the execution of previously constructed distributed applications comprising multiple nodes connected by a network. Each node executes software components, and these components interact through messages. In such applications we are interested in detecting events and patterns of events. In this section we define the main components of the event model: atomic events, messages, event patterns, and pattern detection using DFA.

In this investigation, for simplicity, we will consider only method calls as atomic events. Here the call does not entail execution, the execution of a method is other type of event. Atomic events occur in specific machines within distributed applications, and event detection occurs immediately in that machine. To allow an event to be detected by remote machines in a network, a message must be sent by the machine in which the event occurred to the other machines in the distributed application. The message does not arrive immediately; it may be delayed owing to network and infrastructure latency. The event model assumes that all atomic events generate and broadcast a message; however, we will show that in the implementation only messages concerning events that are of interest for event constructs are broadcast. Messages will transport context information about the originating event. Finally, the order of messages is not granted and depending on network conditions different nodes could see messages in different orders.

Patterns of events are sequences of atomic events that can be described using a Deterministic Finite Automaton (DFA). In the simplest version of the

automaton that we will consider, each transition corresponds to the detection of a particular atomic event. Although the only atomic event that we consider is the call to a method, we can augment the expressive power of the automaton through guard predicates. A guard predicate is a Boolean expression over the context information of an event. Once a specific sequence of events is detected, concrete actions may be executed either in parallel with, before, after, or during any step of an event pattern. Programs written in our language can therefore alter the behavior of the base application.

3.2 Event Classes, instantiation, and synchronous/asynchronous executions of reactions

Before describing the concrete syntax of the language, we introduce the concept of *Event Classes*. These are syntactical units where the automaton to detect event patterns and the reactions to those event patterns are coded. Once an event class is defined, at runtime, it will generate an unique instance of the class on each machine participating in the distributed application (i.e., the default instantiation policy for event classes is Singleton). Such an instance will have embedded one instance of the automaton and will detect events as they arrive at the host where the monitor is deployed. By default, reactions occur asynchronously; thus, each time an event occurs, the message containing the event information will be multicast, and the originating thread will continue to the next instruction in the program. However, a programmer of event classes may decide to process events synchronously, thus making the originating thread to wait for the result of the reaction.

3.3 Syntax and language design

The main abstraction of the language is the denoted event class, which defines objects that monitor the occurrence of event patterns in a distributed application and react accordingly. In the following section, we describe our language in the Extended Backus-Naur Form shown in Figure 1.

3.4 Atomic events

In *EKETAL* the minimal conceptual units are the events. The events that can be manipulated with *EKETAL* are method calls in a Java application. To manipulate these events and react to them, the language allows them to be intercepted using an event definition. The events are defined inside event classes. Each event definition is started with the keyword `event`, followed by an identifier `EId` and a list of parameters. Next an event predicate is added, to identify events that are captured by this event definition (see the non-terminal `EvDcl`).

```
// Event class
Ec              ::= eventclass Id '{' {Decl} '}'
Decl            ::= JVarD | EvDecl | Aut | Rc | MSig
// Atomic event
EvDcl           ::= event EId({Par}):Ep
// Event predicates
Ep              ::= call(ESig) | EId({Par})
                  | host(Group) | on(Group[, Select])
                  | causal({Ep}) | args({Arg})
                  | eq(JExp, JExp) | if(JExp)
                  | Ep || Ep | Ep && Ep | !Ep
Group           ::= { Hosts }
Hosts           ::= localhost | eventhost | "Ip:Port"
                  | GroupId
GroupId         ::= String
Select          ::= JClass
// Complex events: Automata definition
Aut             ::= automaton Id({Par})'{'{Step}'}'
Step            ::= [StateType] Id: TDef
TDef            ::= Ep [→Id ] | TDef || TDef
StateType       ::= start | end
// Reactions
Rc              ::= [syncex] reaction Id Pos [Id.]Id({Par}) '{' {Body} '}'
Pos             ::= before | around | after
Body            ::= JStmt | addGroup(Group) | removeGroup(Group)
// Standard rules (intensionally defined)
MSig, FSig, ESig ::= // method, field signatures (AspectJ-style)
Arg, Par        ::= // argument, parameter expressions (AspectJ-style)
```

Fig. 1: EKETAL language

3.5 Event predicates and causality

This language supports event-oriented programming in a distributed context, where a predicate can monitor events in different hosts. The grammar shown in Figure 1 gives the essential elements of the definitions of predicates in the *EKETAL* language. Predicates can be composed by a terminal `call (ESig)`, which indicates interest in all events that are a call to the set of methods determined by the `Esig`. `Esig` is a non-terminal that represents the signatures of the methods using a syntax similar to that proposed in AspectJ [14]. `host(Group)` is a terminal that defines the group of hosts where an event originated from, i.e, where the `call` occurred. `on(Group)` defines the group of hosts in which reactions are going to be executed. `causal({Ep})` creates a predicate that is true if: `Ep` is true and the event that triggers the evaluation of the predicate is causally related with the last event that caused the atomata to change its state. Note that causality relation is only valid in the context of an event pattern (sequence of events) defined by a DFA. If the event is the first to be evaluated in the automaton, the evaluation of causality is always true. `args({Arg})` defines a group of calls to

methods with a parameter signature specified by the non-terminal `Arg`, which is a parameter expression as those proposed in AspectJ.

`eq(JExp, JExp)` makes an equivalence comparison using Java expressions. `if(JExp)` in the same manner as the conditional if in Java. `Group` defines a group of hosts. The language allows the following terms to be used: `localhost` (the location that hosts the object that monitors events), `eventhost` (the host where an event is executed), `Ip:port` (a specific host), `GroupId`, to refers to groups. Groups can also be referenced by their name, and named groups are dynamically managed within a reaction by adding and removing hosts. The `Ep` predicates can be combined using logical operators such negation, conjunction, and disjunction.

3.6 Complex sequences of events and reactions

The language models complex sequences of events through representation of a deterministic finite automaton with guards. The automaton `Aut` has a name `Id` and a list of parameters `MSig` that follow the same conventions as a method signature in Java. The automaton comprises a set of states with respective transitions (`Step`), and each state has an id tag and may have an event type that indicates whether it is in the initial or final state (`start | end`). It, also defines one or more transitions linked together by the terminal `||`. Each transition defines a named atomic event or a predicate of events, as well as the destination state.

Reactions are activated when a transition's predicate is satisfied. A reaction (non-terminal symbol `Rc`) is defined as follows: the reserved word `reaction` is used; followed by an identifier for the reaction `Id`; a position `Pos` that indicates if the reaction will occur before, after or instead of an event; an event indicator or the state of a machine in which it will react; and a body `Body` built with Java declarations. By default, a reaction is executed in a asynchronous manner, which means that the application does not wait until the reaction is completed to return to its original behavior or to execute the next reaction. A programmer can also select a synchronous reaction by marking the reaction with the reserved word `syncex`.

4 Compiler implementation

To design and implement the compiler we decided to use AspectJ's infrastructure to address the problem of program instrumentation [3]. So, our compiler will generate AspectJ's code augmented with code to support events, distribution, concurrency and automata. Then it will use AspectJ's weaver to weave the event class definitions with the base application code, generating bytecode that is fully compatible with the Java Virtual Machine.

[3] Note for reviewers: The code of the compiler is available at: https://github.com/unicesi/eketal.

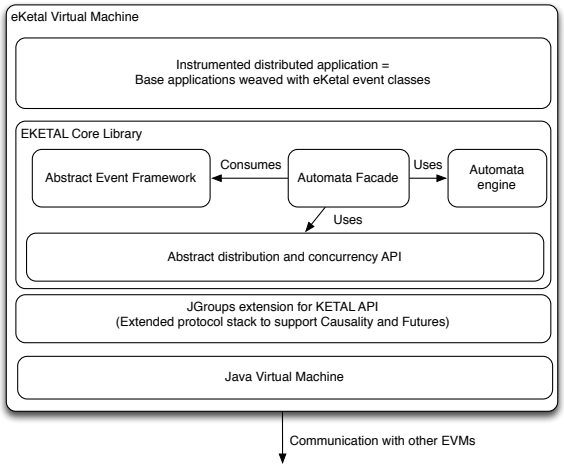


Fig. 2: Architecture of the kernel event-based library

4.1 Runtime Architecture of EKETAL Virtual Machine

The EKETAL Virtual Machine (EVM) operates by a very simple metaphor. First, an event is detected in the base application (the application being instrumented). Then the event is encapsulated in a message and augmented with context information. Then, the event is multicast to all nodes in a distributed application. Finally, the events are consumed and processed by an EVM instance deployed on each node.

Figure 2 presents a layered architecture for the EVM. The top layer corresponds to the application layer and contains the base application weaved with the event classes that define the monitoring and dynamic instrumentation intents of the programmer. Te second layer presents the EKETAL core library. The main components of the library are: the `Abstract Event Framework`, `Automata Facade`, `Automata Engine`, and `Distribution Layer`. The `Abstract Event Framework` defines interfaces to allow developers to create atomic events that are consumable by the automata defined. The `Automata Facade` provides abstractions to manipulate directly the definition and execution of an automaton. The `Automata Engine` is in charge of processing actions over the automata, according to the automata definition. Currently, we are using the automata library provide by Anders Møller et al. [18] as our automata engine.

The `Distribution Layer` provides the main abstractions to distribute event messages and listen to event messages sent by other nodes. The layer provides abstractions and interfaces for a distributed architecture (*i.e.*, with no centralized component). The current implementation of the transport layer is based on group communication using JGroups [13]. To handle the causality predicates and causal ordering of messages (see the discussion of vector clocks and causality in [17]), we have developed two protocols that are configurable on the JGroups protocol

stack. Finally the Java virtual machine supports the implementation of all the components of the EVM.

4.2 Compiler Architecture

The EKETAL's compiler is constructed using the XText framework. From the grammar, the XText framework generates the parser, the linker, the type checker and the compiler. The tool also generates an editor for the eclipse platform. The generated infrastructure was extended with customized code generators in order to create AspectJ code, and the event, automata, and distribution infrastructure.

In order to generate bytecode compatible with the JVM the compiler of EKETAL generates AspectJ code and then uses the AspectJ's weaver. The EKETAL's compiler reads an EKETAL source code and generates AspectJ code. Then, the AspectJ weaver takes the generated code and the base application (the aplication to be instrumented) and weaves them together. The process output is bytecode for the Java Virtual Machine (JVM). Using AspectJ's infrastructure to leverage EKETAL compiler allow us to provide dynamic weaving, thus, avoiding the need of having the source code of the application to be instrumented.

4.3 Atomic event detection and message broadcasting

When implementing the compiler we need to take care of two problems. The first problem is how to detect, on the base application, the specific atomic events we are interested in. The second problem is how to distribute and manipulate the events in order to match the event patterns we are interested in. To solve the first problem we decide to use AspectJ's pointcut like constructs, thus EKETAL's inherits part of its syntax for event predicates from AspectJ's pointcut syntax. However, we have to take special care with the *host*, *on* and *causal* predicates. The *host* construct predicates about the localization of the atomic event, e.g., where a method call occurred. Thus, when translating the predicate into AspectJ we may model the construct as an *if* pointcut. For example, the following event predicate:

```
call(point.SetX(..)) && host("Screens Group")
```

will translate into the following pointcut:

```
call(point.SetX(..)) && if(inGroup("Screens Group"))
```

where `inGroup` is a function that returns `true` if the host, where the call occurs, belongs to the *"Screens Group"* group. Then the advice (code that is executed when the pointcut is matched on AspectJ) attached to this pointcut creates an event representation and broadcast a message with the event information to the other EVMs.

With respect to the second problem and particularly in the context of `on` and `causal` predicates, a more complicated situation arise. These constructs

predicate about the context of the event class instance, and the context of the distributed event message. In the first case, the event detection infrastructure has to evaluate the arriving event, in order to decide if the event class is deployed in the group defined by the `on` construct. In the second case, the causal predicate, the event detection infrastructure has to evaluate if the arriving event has a causal relation with the previously matched event. The implementation, in this case is not a simple translation into AspectJ's pointcut expressions. Instead you have to construct first an expression to detect the atomic event and broadcast the atomic event occurrence, and another expression to match and evaluate the distributed events.

Consider for example the following predicate:

```
causal(call(point.SetX(..)) && on("Server Group")),
```

in order to detect the atomic event it will translate into the following pointcut:

```
call(point.SetX(..)) && if(true),
```

this pointcut will then broadcast the corresponding message. However, the compiler will generate also a boolean expression to detect and match the distributed event message:

```
if(atomicEventDefinition.causallyMatches(remoteMessage)
                            && onHost("Server Group"),
```

where `atomicEventDefinition` is an object representing the definition of the atomic event; `causallyMatches` is a boolean method that returns true if the definition matches the atomic event contained in the `remoteMessage` and the event is causally related with the last evaluated event; `onHost` is a boolean method that returns `true` if it is evaluated on a host belonging to the "Server Group" group. Once the implementation of distribution and the matching framework is solved, the rest of the compiler is implemented translating the code into the components of the core library of the EVM.

5 Validation: Debugging and Testing concurrency problems on distributed middleware

In this section we provide qualitative and quantitative evaluation of EKETAL. First, we evaluate EKETAL's runtime performance by means of distributed experiments using Hadoop (Big Data framework implementing map-reduce algorithms). Then, we present a quantitative survey of concurrency and distribution errors reported on open source projects, and show how EKETAL may help to unit-test, debug and correct some of those errors.

5.1 Performance evaluation of EKETAL

To test the performance of EKETAL we have designed two experiments using Hadoop[4]. In the first experiment we instrument the Hadoop framework by means of a simple eventclass that matches a method that may be called infrequently in any host of the distributed application (`setup` method). This experiment will test the overhead of adding EKETAL's runtime to a Java Virtual Machine running Haddop. The second experiment will test the overhead of using EKETAL on methods that are called very frequently in the distributed application (`map` method), generating high network traffic due to event broadcast. Both experiments are run in a cluster of virtual machines on Amazon Web Services (AWS). We compare each experiment with a regular execution of Hadoop (i.e, no instrumentation).

The experiments will run a map-reduce algorithm to count the number of times a word is found in an unstructured data set, e.g., counting the most used words in a search engine. The map phase will create on each node a set of pairs $(word, 1)$, for each time a word appears on the data set. The reduce phase will create a final result where there is only one pair for each word. The original data set is a text file that has 15.443 lines, 144.181 words, and 13.000 distinct words. The `map` method is the method we intercept as a method that is called with high frequency. The experiments will be run on two hardware configurations on Amazon Web Services. First on 2 nodes and then on 3 nodes, each node will have 1 virtual cpu (Intel Xeon 2.5 GHz) with 2 Gb of memory. Each experiment is executed 20 times.

Table 1 shows the average time of execution of the map reduce algorithm over different data sets and the number of events that are broadcast on each experiment. The first line in the table details the execution time of Hadoop with no instrumentation, in this case the execution time is 10 sec. on average. The second line shows the average time of the execution of Hadoop instrumented with EKETAL on a method that is called rarely, this experiment is designed to measure the overhead of EKETAL's distributed infrastructure. The result shows no significant difference on execution time between the execution with low frequency and high frequency methods. In fact Hadoop randomly creates new virtual machines to execute the algorithm, but having only 1 file as input makes the framework to execute most of the time on one node. Thus, the time shown in the tables seems to be related to the delay generated by the EKETALs distribution libraries when they start. When we analyze the results when three nodes are used to run the application, and a bigger data set, the situation is more interesting. Here the execution o Hadoop with no instrumentation takes only 14s, and there is no difference when the experiment is executed with a low frequency method instrumented. When we instrument a high frequency method the average time is 56 sec. We can explain this time because now we have 77.215 events broadcast (from 5 different files as input), and all the machines are detecting those events. However, if we see figure 3 we can see that not all the events are

[4] Hadoop is a Big-Data framework designed for distributed processing of large data sets across clusters of computers using map-reduce algorithms

Experiment: Hadoop with	Nodes	Processor	No. Events	Time Ex.
No intrumentation	2	Xeon, 2 Gb	0	10.541,85
EKETAL (low freq.)	2	Xeon, 2 Gb	1	17.267,8
EKETAL (High freq.)	2	Xeon, 2 Gb	15443	17.243,8
No intrumentation	3	Xeon, 2 Gb	0	14.482,85
EKETAL (low freq.)	3	Xeon, 2 Gb	5	14.283,55
EKETAL (High freq.)	3	Xeon, 2 Gb	77215	56.107,00

Table 1: Average execution time for map reduce experiments with 13000 words using Haddop instrumented with EKETAL.

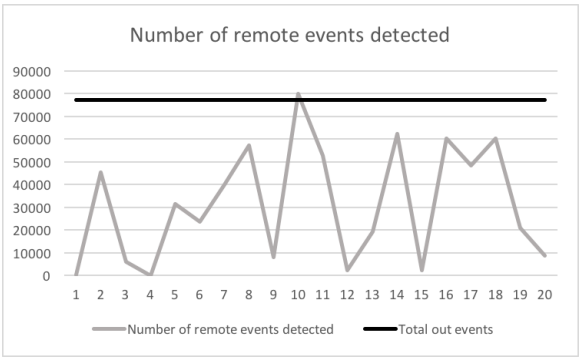
detected remotely, this is because Hadoop do not start the parallel workers at the same time. S,o while they are running there is a window of intersection where workers may detect events originated on each other. The average number of events detected is 31420, which means that in average there are at least two workers that run in parallel. In figure 3 we can see also the distribution of the execution times on three nodes over AWS.

5.2 Debugging and Testing using EKETAL

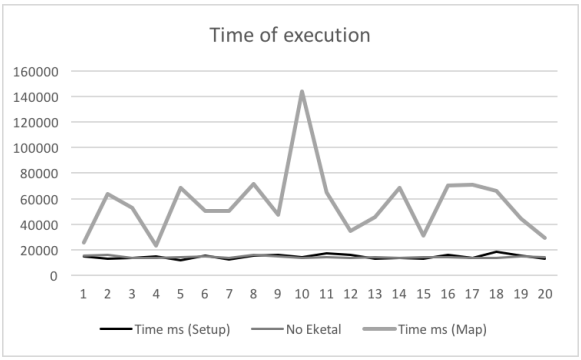
Application	No. Issues	Solved	Open	C&D Open	% C&D Open	C&D All	% C&D All
Active MQ	4947	4390	557	72	12,92%	506	10,22%
Hadoop	5998	3957	2041	229	11,21%	840	14,00%
JBoss Cache	951	833	118	6	5,08%	56	5,88%
Spark	18004	15284	2720	225	8,27%	991	5,06%
Asterix DB	906	348	558	30	5,38%	78	8,61%
Total	30806	24812	5994	562	9,38%	2471	8,02%

Table 2: Characterization of concurrent and distributed (C&D) issues found in distributed applications.

We have investigated the applicability of EKETAL for testing and debugging real world applications. In this section we first present a characterization of issues found in several open source distributed applications, quantifying the number of cases where liveness o data race errors where likely to cause the problem, and where their reproduction (i.e., test case) and their solution depended on enforcing a specific order of events. We then study a concrete example and show how EKETAl may help to unit-test, debug, and correct this type of issues. Table 2 shows data of reported issues on several open source projects implementing distributed applications. The table includes data for Active MQ (A message Broker), Hadoop, JBoss Cache (a distributed and replicated cache), Spark (engine for large scale data processing), and Asterix DB (a scalable, open source



(a)



(b)

Fig. 3: Data for 20 executions of the experiment on 3 nodes on AWS. a) Number of remote events detected by the monitors. b) Time of execution of the experiment

Big Data Management System). The table shows the number of reported issues, how many of them still open, and what percentage of them are likely to be related to concurrent and distribution problems. For example, considering Hadoop, there are 5998 reported issues, 2041 from those issues are still open, and from those 11,21% (229) are related to problems of concurrency and distribution, the percentage is even bigger when we consider that 14% from the total of reported issues are related to concurrency and distribution problems. When we take together the 5 applications analyzed, around 10% of open issues are related to problems of distribution and concurrency. This is not an small problem, considering that the reproduction is, in most of the cases, a non deterministic task, and then, debugging and correction becomes a complex task.

Dataraces in JBoss Cache. In versions 1.3.0.GA – 1.4.0.SP1 of *JBoss Cache* a *datarace* problem was found between the cache process and *eviction*

process [21][5]. This problem persisted for approximately five months before being solved in versions 2.0.0.BETA2 and 2.0.0.GA. This error exemplifies problems that are hard to reproduce and correct in distributed applications. The *eviction* process is in charge of cleaning cache information by removing all stored data that is not used or referenced within a specific period of time. This policy gives storage priority to information most frequently accessed in the cache. To illustrate the *datarace* problem, we assume the existence of two nodes, the first node is the main cache, which stores the information most frequently accessed by nodes in the network, and the second node is a client asking for information. In the cache there is `Eviction thread`, handling the eviction policy, and for each data request the application generates a new `Handler thread`. The data race occurs because an eviction request and a read request for the same data `X` arrive to the cache node at the same time, and the following sequence of events happen in this specific order: the `EvictionThread` wins the race and acquires the lock on `X`, then the `Handler Thread` verifies that `X` is stored in memory ignoring that the `EvictionThread` has placed a lock on X, the `EvictionThread` removes `X`, and after that, the `HandlerThread` invokes the method `get(X)` which replies null due to the fact that X has been removed from the main cache.

In Figure 4, we use the *EKETAL* language to write an event class that implements a monitor to detect the error described above. In the implementation, object `x` is the value we want to store and the fully qualified name (`fqn`) is the location on the cache where the object must be stored or from where the object reference must be read. Lines 2-13 define the atomic events that lead to the *data race*. Lines 2-3 describe the event `dataRequest` with two parameter types: `Fqn` type and `Object` type. This event captures any call to the `get` method, which occurs when `X` is requested by any node in the network. Note that the values of the fully qualified name and the object value are bound to the values defined by this first event. Lines 4-6 describe an event that receives the same parameters as the previous event and that takes place in the local host. This event captures any call to the method `evict`, which is responsible for removing the indicated object from the main cache under the *eviction* policy.

Lines 7 and 8 define an event that executes the eviction action: obtains the lock necessary to modify the cache's memory structure, and removes the node `name`. Lines 9-11 describe the occurrence of the low level method to get the data `X` from the main cache. Lines 13-18 define the automaton that detects the *data race*, which contains four states and four transitions. In the initial state `init`, when the `dataRequest` event occurs, it advances to the `waitForEvict` state. Then, if the event `dataRetievalAction` occurs, the automaton returns to the initial state, because the data race can not happen. Alternatively, if the `evictRequest` event arrives, it moves to the state denoted as `waitForEvictToWinRace`. At this point, two situations can occur. The first can occur when the event `evictAction` is executed, which moves automaton to the `dataRace` state. The second situation takes the automaton to the `init` state. When the automaton reaches the `dataRace` state, a data race has happened.

[5] The id of the error in JBoss Cache's JIRA repository is: JBCACHE-814.

```
1  eventclass DataraceTest{
2   event dataRequest(Fqn name, Object x):
3       call(* TreeCache.get(name, x));
4   event evictRequest(Fqn name, Object x):
5       call(* EvictionPolicy.evict(name)) && host(localhost);
6   event evictAction(Fqn name):
7       call(* TreeCache._evict(name)) && host(localhost);
8   event dataRetrievalAction(Fqn name, Object x):
9       call(* TreeCache._get(name) && host(localhost));
10
11  automaton dataraceDetector(Fqn f1, Object x){
12      start init: dataRequest(f1, x) > waitForEvict;
13      waitForEvict:(evictRequest(f1, x) > waitForEvictToWinRace) ||
14                   (dataRetrievalAction(f1, x) > init);
15      waitForEvictToWinRace: (evictAction(f1) > dataRace) ||
16                   dataRetrievalAction(f1, x) > init;;
17      end dataRace;
18  }
19
20  reaction beforeDatarace
21      before dataraceDetector.dataRace(Fqn f1, Object x){
22          //Reaction to deadlock}}
```

Fig. 4: Data race detection with EKETAL

6 Conclusions

This study proposes an event model and the corresponding design for an event-oriented programming language, its virtual machine, and the compiler to address debugging and testing in distributed and concurrent applications. The language allows the declaration, execution, detection, and coordination of complex event patterns in distributed systems. It also introduces explicit support for the detection of complex sequences of events using finite automata, guard support, and support for futures and for synchronous and asynchronous coordination of reactions and the base program.

We evaluate EKETAL's runtime performance by means of distributed experiments using Hadoop. By applying the language, we demonstrate that the debugging and maintenance problems of *liveness* and *dataraces* found in distributed applications can be addressed. In particular, we show how breakpoints can be modeled by complex sequences of events and demonstrate the creation of deterministic test scenarios for concurrent and distributed applications. However, the mechanism has several limitations that prevent modeling, for example, patterns of events that include temporary conditions or patterns that include parameters reliant on event counts. In future work, we will investigate the adoption of other abstractions such as *push-down automata* and temporal logic to extend the functionality and scope of our tool.

References

1. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.
2. L. D. Benavides Navarro, R. Douence, and M. Südholt. Debugging and testing middleware with aspect-based control-flow and causal patterns. In *In Proc. of the 9th Int. Middleware Conference*, Leuven, Belgium, Dec. 2008. Springer-Verlag.
3. N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for c#. In *Proc. of the 16th European Conference on Object-Oriented Programming*, ECOOP '02, pages 415–440, London, UK, UK, 2002. Springer-Verlag.
4. M. Butler and S. Hallerstede. The rodin formal modeling tool. In *Proc. of the 2007th Int. Conference on Formal Methods in Industry*, FACS-FMI'07, pages 2–2, Swinton, UK, UK, 2007. British Computer Society.
5. F. Chen and G. Roşu. Mop: An efficient and generic runtime verification framework. *SIGPLAN Not.*, 42(10):569–588, Oct. 2007.
6. C. Colombo, G. J. Pace, and G. Schneider. Larva — safer monitoring of real-time java programs (tool paper). In *Seventh IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, pages 33–37. IEEE Computer Society, November 2009.
7. A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards expressive publish/subscribe systems. In *Proc. of the 10th Int. Conf. on Advances in Database Technology*, EDBT'06, pages 627–644, Berlin, Heidelberg, 2006. Springer-Verlag.
8. M. Eriksen. Effective scala, 2012. Available: http://twitter.github.io/effectivescala/.
9. P. Eugster and K. Jayaram. Eventjava: An extension of java for event correlation. In S. Drossopoulou, editor, *ECOOP 2009 – Object-Oriented Prog.*, volume 5653 of *Lec. Notes in Computer Science*, pages 570–594. Springer Berlin Heidelberg, 2009.
10. C. Fournet and G. Gonthier. The reflexive cham and the join-calculus. In *Proc. of the 23rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, POPL '96, pages 372–385, New York, NY, USA, 1996. ACM.
11. W. Inc. Whatsapp open source, 2016. Available: https://www.whatsapp.com/opensource/.
12. K. R. Jayaram and P. Eugster. Scalable efficient composite event detection. In *Proc. of the 12th Int. Conf. on Coordination Models and Languages*, COORDINATION'10, pages 168–182, Berlin, Heidelberg, 2010. Springer-Verlag.
13. Jgroups home page. latest visit on June 2015, 2011.
14. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *Proc. of the 15th European Conf. on Object-Oriented Prog.*, ECOOP '01, pages 327–353, London, UK, 2001. Springer-Verlag.
15. P. Leger, E. Tanter, and H. Fukuda. An expressive stateful aspect language. *Sci. Comput. Program.*, 102(C):108–141, May 2015.
16. Q. Luo and G. Rosu. EnforceMOP: a runtime property enforcement system for multithreaded programs. In *the 2013 International Symposium*, pages 156–166, New York, New York, USA, 2013. ACM Press.
17. F. Mattern. Virtual time and global states of distributed systems. In *Proc. of the Int. Workshop on Parallel and distributed Algorithms*, Chateau de Bonas, France, October 1988.
18. A. Møller. dk.brics.automaton – finite-state automata and regular expressions for Java. latest visit on May 2011, 2010.

19. V. Rivera and N. Cataño. Translating event-b to jml-specified java programs. In *Proc. of the 29th Annual ACM Symp. on Applied Computing*, SAC '14, pages 1264–1271, New York, NY, USA, 2014. ACM.
20. K. Serebryany and T. Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proc. of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, New York, NY, USA, 2009. ACM.
21. J. C. TreeCache. A structured, replicated, transactional cache. user documentation., 2013. Available: http://docs.jboss.org/jbosscache/1.4.0/TreeCache/.
22. Y. Zhuang and S. Chiba. Method slots: Supporting methods, events, and advices by a single language construct. In *Proc. of the 12th Annual Int. Conf. on Aspect-oriented Software Development*, AOSD '13, pages 197–208, N.Y., USA, 2013. ACM.