Debugging and Testing Middleware with Aspect-Based Control-Flow and Causal Patterns*

Luis Daniel Benavides Navarro, Rémi Douence, and Mario Südholt

OBASCO project; EMN-INRIA, LINA
Dépt. Informatique, École des Mines de Nantes
4 rue Alfred Kastler, 44307 Nantes cédex 3, France
{lbenavid,douence,sudholt}@emn.fr

Abstract. Many tasks that involve the dynamic manipulation of middleware and large-scale distributed applications, such as debugging and testing, require the monitoring of intricate relationships of execution events that trigger modifications to the executing system. Furthermore, events often are of interest only if they occur as part of specific execution traces and not all possible non-deterministic interleavings of events in these traces. Current techniques and tools for the definition of such manipulations provide only very limited support for such event relationships and do not allow to concisely define restrictions on the interleaving of events.

In this paper, we argue for the use of aspect-based high-level programming abstractions for the definition of relationships between execution events of distributed systems and the control of non-deterministic interleavings of events. Concretely, we provide the following contributions: we (i) motivate that such abstractions improve on current debugging and testing methods for middleware, (ii) introduce corresponding language support for pointcuts and advice defined in terms of causal event sequences by extending an existing aspect-oriented system for the dynamic manipulation of distributed systems, and (iii) evaluate our approach in the context of the debugging and testing of Java-based middlewares, in particular, JBoss Cache for replicated caching.

1 Introduction

Many tasks that involve the dynamic manipulation of middleware and large-scale distributed applications, such as debugging and testing, require the monitoring of intricate relationships of execution events that trigger modifications to the executing system. Such relationships, which often include events occuring on different hosts, have to be defined declaratively as well as monitored and modified efficiently. Consider, for instance, coherency of replicated data under transactional control in middleware cache infrastructures, such as JBoss Cache: in this

* Work partially supported by AOSD-Europe, the European Network of Excellence in AOSD (www.aosd-europe.net).

V. Issarny and R. Schantz (Eds.): Middleware 2008, LNCS 5346, pp. 183–202, 2008.

case, the correctness of sequences of events corresponding to executions of two-phase-commit protocols involving multiple machines has to be checked. Furthermore, execution events of a distributed system frequently are of interest only if they occur as part of specific 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. The definition of reproducible test cases, for instance, frequently requires constraints to be imposed on non-deterministic executions.

Several approaches to define such relationships among and constraints on events in distributed systems have been proposed. Such approaches include, for example, causal event relationships based on logical clocks [1, 13, 17], data path expressions for concurrent programs [23], and control-flow based event relationships [18]. However, such declarative means for the definition of event relationships have not been integrated into mainstream middlewares and corresponding support in current tools for the debugging and testing of distributed infrastructures is very limited. Intricate relationships between distributed events and restrictions on the interleavings of concurrent events can be directly defined in current execution environments only in terms of conditions on the execution state of individual hosts. Hence, relationships involving multiple hosts have to be expressed using complex encodings that are difficult to understand, to maintain, and result in inefficient event monitoring and execution of modifications.

In this paper, we argue for the use of high-level abstractions for the definition of relationships between execution events of distributed systems, their modification and the control of non-deterministic interleavings of events. Concretely, we provide three contributions. First, we motivate that such mechanisms improve on current debugging and testing methods for distributed systems, in particular, real-world middleware infrastructures (Sec. 2). Second, we introduce corresponding aspect-based programming language support that provides declarative means to monitor and modify causal sequences of events in pointcuts and advice. We present suitable language support (Sec. 3) and a corresponding implementation (Sec. 4) in terms of an extension of the AWED language and system [2, 4] for the dynamic manipulation of distributed systems using distributed aspects. Third, we evaluate our approach in Sec. 5 in the context Java-based middlewares, in particular, for debugging and testing of JBoss Cache, a Java-based middleware for replicated caching, and ActiveMQ, the Apache message broker. We also show how current best practices for the debugging and testing of distributed systems can be improved using our approach in a practical and efficient manner. Related work is discussed in Sec. 6 and a conclusion given in Sec. 7.

A copy of the code, the benchmarks and evaluations in the context of JBossCache and ActiveMQ can be found at [2].

2 Motivation

In this paper, we argue for the use of sophisticated relationships between events to be used to monitor and manipulate middleware and distributed infrastructures. We claim, in particular, that control-flow based relationships, sequence

relationships and events that are causally-connected, *e.g.*, with respect to a notion of logical time, are crucial in this context. In this section, we motivate these claims for typical debugging and testing tasks of distributed infrastructures.

2.1 Expressive Breakpoints for Distributed Debugging

Current tools for distributed debugging, such as Eclipse and the Distributed Debugging Tool [12, 26], apply debugging techniques for sequential programs to distributed applications. Such tools almost always employ a centralized debugging component that coordinates execution of independent local debuggers that only support breakpoints in terms of local entities (*e.g.*, updates of local objects, local files, etc.). The distributed debugger can match local breakpoints in different machines and control the execution by, *e.g.*, stopping it and inspecting the local state of different machines. However, this kind of tools has not been widely adopted by developers, mainly because they do offer only small added value over the use of sequential debuggers on a per-machine basis.

We argue that there are two major reasons for this lack of added value:

- Lack of means for the expressive definitions of distributed breakpoints involving, in particular, control flow and sequence relationships between distributed execution events.
- Lack of means to handle non-determinism in distributed and concurrent applications.

In the following, we consider three basic debugging scenarios that frequently occur in middlewares to illustrate these issues involving control-flow relationships and non-deterministic relationships among events, especially ones involving causally-connected events (thus effectively extending discussions in recent work on distributed debugging [18, 20]).

Debugging control flow. As a first example, consider a distributed application that uses synchronous remote method invocation (*e.g.*, Java RMI) for communication between three different hosts. A developer may be interested in setting a line breakpoint in one host, `H` say, that is triggered only in the dynamic extent of a (previous) method call occuring on another host `G`. Note that such debugging scenarios are based on (typically implicit) specifications of correct program behavior. *e.g.*, that an erroneous execution path is characterized by the sequence of calls `G;H` on the mentioned hosts where `H` occurs before the call to `G` returns. Using current tools, the developer has three options:

- She can apply a breakpoint to the method called on host `G` and once this match is triggered she can, at runtime, add the line breakpoint at `H`. However, in this case all subsequent occurrences of the second breakpoint are matched: identifying a specific call of interest can be very difficult.
- The programmer could pollute the original code with state information to track the necessary control flow dependencies (*i.e.*, store state information that then has to be suitably forwarded to the other hosts) and match the specific breakpoint in `H`.

– The programmer could add a breakpoint directly on the execution of `H`, match the corresponding breakpoint there without taking into account the originating control flow and decide manually what to do at each match.

Using (formal or informal) reasoning mechanisms, all of these options could be proven to correctly identify the erroneous path with respect to the specifications above. However, clearly none of these situations is acceptable, because they are tedious to implement and are highly error prone. All three represent common practice with current debuggers for distributed middleware and applications, though.

Debugging non-deterministic event relations. Events that may occur concurrently and that should trigger debugging operations only if they are interleaved in specific ways further complicate matters. Debugging of replicated caching infrastructures, for example, may involve replication actions that originate from the same transaction but are triggered asynchronously (e.g. as part of a two phase-commit protocol). Errors often depend in this case on the order in which the replication actions are applied but the decision, as part of a debugging action, whether two actions occur in the relevant order is difficult to take if debugging processes (as is often the case) may introduce arbitrary delays in the observation of events.

Since current debugging tools do not provide abstractions to concisely express such cases, programmers once again have to resort to manually encode and interpret distributed state by applying one of the three options introduced above. This approach becomes, however, rapidly unmanageable if many events and many hosts are involved.

Often such debugging tasks can be much facilitated by ensuring that occurrences of events obey strict ordering requirements, possibly imposing deterministic sequences of events in a previously non-deterministic systems. This is useful, in particular, in order to systematically explore possible erroneous traces. Once again current debuggers do not support such facilities, but have to resort to encodings of distributed state. Extending previous work [14, 18, 20, 23] that has highlighted casual relationships as a means to remedy this problem, our approach seamlessly integrates notions of causality with expressive control-flow based event relationships.

2.2 Test-Driven Development

Current techniques for the test-driven development for distributed applications are also limited by a lack of support for the expression of distributed event relationships. Distributed unit test cases, in particular, are almost always implemented by means of sequential abstractions that test conditions of distributed concerns on the local state of individual machines. For example, test cases related to replication in JBoss Cache [15] frequently use a seemingly intuitive testing scenario: a test case is defined in terms of two cache instances, such that after an operation on a source cache, the state of the second cache can be tested to

compare the new and old versions. This idiom seems obvious and simple; however, it does not allow to take into account, for example, the communication behavior, such as sequences of intermediate synchronous or asynchronous calls, which obviously may strongly interfere with the cache behavior. Consequently, the definition of reproducible test cases are subject to the same restrictions as discussed above, for example, if reproducibility depends on specific interleavings of a set of concurrent events being tested (that are part of a potentially much larger set of possible interleavings).

3 Language Support

In this section we propose a language to support manipulations and evolutions of distributed applications. It is based on the AWED system (Aspect With Explicit Distribution [4, 6]): that explicitly supports monitoring of sequences of distributed execution events that trigger dynamic modifications. This enables us to concisely express different debugging scenarios involving control-flow and sequence-like relationships between events. Furthermore, we introduce in this paper an extension of AWED in order to support causally-related events and causal communications (based on an event reordering mechanisms).

3.1 The AWED Language

Aspect Oriented Programming supports separation of concerns. An aspect modifies a base application: its pointcut specifies points of interest (*i.e.*, events) in the base application execution and its advice specifies a piece of code to be executed before, after, or instead of such a point of interest. In this paper, a pointcut can denote a single event (*e.g.*, a method call) or a sophisticated sequence of events. The base application and the aspect are woven into a single application where the aspect monitors the base program execution and triggers its advice.

AWED supports AOP in a distributed context. In particular, a pointcut can monitor events on several hosts. A sequence of events can involve different hosts. An advice can be executed remotely, synchronously or asynchronously to the base execution. Furthermore, an aspect can be deployed on a group of hosts.

The grammar shown in Fig. 1 shows the essentials of pointcut definitions in the AWED language (the full language definition can be found in [6]). The pointcut language allows matching of method calls (terminal `call`), nested calls (`cflow` means control-flow) and arbitrary (regular) sequences of method calls (non-terminal *Seq*). The constructors `host` and `on` specify (groups of) hosts where a pointcut is matched (or where an advice is executed). The constructors `target` and `args` bind values (such as the receiver or the arguments of a method call) to variables. This enables values to be passed from a matching execution event to the corresponding advice. Pointcuts can be composed using logical operators (union, intersection and complement). Sequences (*Seq*) are defined in terms of transitions of non-deterministic finite-state automata. An automaton is a set of transitions *Step*. Each transition has a label *id* and its pointcut *Pc*

```
// Pointcuts
Pc      ::= call(MSig)  | cflow(Pc) | Seq
          | host(Group) | on({ Hosts })
          | target({Type}) | args({Arg})
          | Pc || Pc  | Pc && Pc  | !Pc
Seq     ::= Id: seq({Step})  | step(Id,Id)
Step    ::= Id: Pc → Target
Target  ::= Id  ||  ...  ||  Id
Hosts   ::= localhost  | jphost  | "Ip:Port"  | GroupId
```

Fig. 1. The AWED language (excerpts)

non-deterministically leads to a set of *Id*. The constructor `step` identifies the transition in the automaton that should trigger advice.

3.2 Distributed Debugging with AWED

AWED can be applied to debug intricate relationships between execution events. It generalizes previous approaches to the debugging of control-flow based relationships between events. In this subsection we show how the original AWED model allows to handle debugging problems expressed in terms of control-flow-based and arbitrary sequence-based relationships between distributed events.

Distributed control flow. Sequences of calls that are nested within each other's control flow can be defined using the `cflow` pointcut constructor. Extending Nishizawa's *et al.* [22] work, AWED supports control-flow pointcuts over distributed executions taking into account Java's thread model as well: it enables matching of sequences of events that originate in local threads, span threads spawned at remote locations, and spawned child threads. The control-flow model is also transparent regarding synchronous and asynchronous communication contexts.

As an example consider testing and debugging of JBoss Cache as presented in the motivation section. A concrete problem of the two-phase commit protocol consists in ensuring that remote calls to `prepare` methods are always triggered by a corresponding call at a local cache site. A remote call that has not been appropriately triggered can be caught by the following pointcut:

```
!cflow(call(* Transaction.prepare(..)) && host("source"))
&& call(* Cache.remotePrepare(..)) && host("target")
```

This pointcut matches all the calls to the `remotePrepare` method on hosts belonging to the host (or host group) `target` that are not in the distributed control flow of calls occurring at source hosts. Hence, a simple pointcut definition can address the complexity of a distributed control flow breakpoint. Such control-flow relationships for debugging have already been studied, *e.g.*, as part of Li's work [18] for distributed (CORBA and COM) component-based systems and Chern and De Volder's work on sequential control-flow based breakpoints [9]: we

extend such approaches by supporting the notion of control flow in the presence of asynchronous and synchronous method calls.

Distributed sequences of events. As introduced above, AWED supports pointcuts over sequences of execution events, *e.g.*, sequences of calls that do not have to be nested into other calls of the sequence. Hence, such sequences allow the definition of more general event-based contexts than the control-flow based event sequences considered above.

In the context of the debugging of JBoss Cache, for example, a very frequent requirement consists in the definition of contexts depending on the activation state of the cache. Concretely, one may want to identify remote `put` operations (which introduce data in the cache) that occur after the local cache has been initialized and before it has been stopped. A corresponding pointcut can be specified in AWED as follows:

```
a1 : seq(start > t1,
         t1: call(* Cache.start(..)) && host(localhost)> t2 || t3,
         t2: call(* Cache.put(..)) && !host(localhost) > t2 ||t3,
         t3: call(* Cache.stop(..)) && host(localhost) > t1)
       && step(a1,t2)
```

This pointcut defines an automaton named `a1` having three transitions `t1`, `t2` and `t3`: once started, `put` operations can occur or the cache can be stopped. Note that the `start` and `stop` operations of the cache are matched on the local host, while the `put` operations must not occur on the local host. The term `step(a1, t2)` allows an advice to be triggered relative to a specific transition `t2` of the automaton. At the first line `start > t1` defines that the initial transition is `t1`. The expression `t1: pointcutDef > t2 || t3` is interpreted as follows: if `pointcutDef` matches the current event, then the automaton is now ready to accept an execution event as defined by `t2` or `t3`. Figure 2 shows the graphical interpretation of the defined automaton.

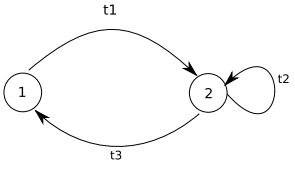

Fig. 2. Graphical representation of a start-action(s)-stop automaton

The expressive power of our approach is mainly determined by the expressivity of our pointcut language. AWED basically provides regular pointcuts. An extension by guards on transitions of the corresponding finite-state machines, thus providing a turing-complete pointcut language, is however unproblematic (and is provided as part of the existing implementation). This feature would also allow to directly characterize concurrent and timed events. By explicitly providing regular pointcuts, existing analysis techniques of, *e.g.*, deadlocks using model checking of distributed and concurrent systems, should be applicable. This is, however, subject of future work.

A second element determining the power of our approach is the granularity of events that can be referred to by pointcuts. We have restricted the pointcut language deliberately to method calls: a more fine-grained event model that would

allow, e.g., to refer to the evaluation of subexpressions of arithmetical expressions (that are supported by some aspect approaches) could incur considerable execution overhead and are less relevant for the debugging of middlewares.

The case for causality relationships. Sequence pointcuts in AWED do not guard against problems of the underlying communication network, in particular concerning message delivery such as inversion of sent messages due to random delays in message transmission. The previous sequence pointcut involving `start`, `put` and `stop` on JBoss Cache events is unproblematic in this respect since message inversions resulting in `put` operations outside the ordinary operating conditions of cache can be easily filtered out by additional pointcuts if necessary. In other cases, *e.g.*, inversion of bank deposits and withdrawals, such problems would however wreak havoc.

Generally, AWED's automata-based pointcuts are therefore subject to two problems:

- They may not match valid sequences of events that happen to arrive in the wrong order at the host where the sequence is to be matched.
- They may match wrong sequences that stem from events that occur at different hosts in the wrong order but whose order has been inverted, *e.g.*, because of message delays, at the host where the sequence is matched.

An AWED developer has to take care in order to avoid these problems: either by the careful definition of pointcuts and manual synchronization of distributed executions or by ensuring that additional constraints on the base application's semantics exclude them. The next subsection proposes new language constructs to enable pointcuts to directly support causality relationships and ordering constraints of messages.

3.3 AWED with Causal Pointcuts

Much research work has been done on orderings of distributed events starting with Lamport's landmark paper [17] on the use of *logical time*. In particular, vector clocks [19] can be used to enforce causal relations between events and implement causal communication by reordering events. We now show how we have integrated these notions into AWED.

Causal sequences without reordering. To extend AWED with causal information, without including reordering of messages, we have introduced a new sequence constructor `seqCausal` and two transition modifiers, `causal` and `conc`,

// Pointcuts
Seq ::= *Id*: *SeqCons*({*Step*}) | `step`(*Id*,*Id*)
SeqCons ::= `seq` | `seqCausal` | `seqCausalOrder`
Step ::= [[!]`causal` | `conc`]*Id*: *Pc* → *Target*

Fig. 3. AWED with causal pointcuts

see Fig. 3. The two modifiers respectively ensure that the labelled transition is causally related to or concurrently executed with respect to the transitions leading to the start state of the labelled transition. The constructor `seqCausal` is syntactic sugar for sequence pointcuts whose transitions are by default labelled as `causal` unless they have been explicitly declared using `conc` to execute concurrently.

As an example let us consider the following pointcut definition:

a1 : **seqCausal**(**causal** s1: **call**(∗ Cache.prepare(..)) && **host**("source") > s2,
conc s2: **call**(∗ Cache.commit(..)) && **host**("target") > s1)
&& **step**(a1, s2)

This sequence matches a `prepare` event in a JBoss Cache transaction, followed by a `commit` only if it is *not* causally related to the `prepare` event. Then the following `prepare` event is matched only if it is causally related to the previous matched `commit` event. This pointcut can therefore be used to test for unexpected calls to commit methods. As we show in the evaluation section, Sec. 5, this pointcut is useful to test for a real bug that affected the JBoss Cache infrastructure.

Causal pointcuts with reordering. Causal pointcuts without reordering only enforce that causally-related events are matched but they do not ensure all sequences will be matched.

To resolve this second problem, we harness the property — already demonstrated by Lamport's totally ordered broadcast operation [17] — that logical time values cannot only be used to test for causality relationships but that they also support the reordering of messages that arrive at a host in the wrong order. To allow reordering according to causal relationships, we have extended AWED with a third sequence pointcut constructor, `seqCausalOrder` that ensures that all causal relations are matched by, if necessary reordering, incoming events. Its semantics ensures that each event is delayed to wait for the event that precedes it causally.

As a concrete example, the following pointcut can be used to ensure that `commit` operations are correctly interleaved with `prepare` operations:

a1 : **seqCausalOrder**(
t1: **call**(∗ Cache.prepare(..)) && **host**("source") > t2 || t3,
t2: **call**(∗ Cache.commit(..)) && **host**("target") > t1,
t3: **call**(∗ Cache.prepare(..)) && **host**("source"))
&& **step**(a1, t3)

Indeed, a cache web repeats sequences of `prepare commit`. So, two `prepare` should never occur in a row (transition `t3`): an error should be reported in this case. In order to prevent reporting of spurious errors (*e.g.*, when a `commit` occurs before `prepare` but is monitored after it) the messages must be ordered as specified by `seqCausalOrder`.

Note that this construct requires a larger overhead than the one without reordering. In particular with the previous construct the events are consumed as soon as they arrive, and causality is only an additional test defined by the

`causal` and `conc` labels. In the case of causally ordered sequences, messages are delayed and processed only once all the causally preceding messages are received. The `causal` and `conc` labels are automatically supported in the totally ordered construct (they do not pose an additional overhead).

4 Implementation

In this section, we present how distributed aspects with support for causal events and message reordering have been implemented by extending the non-causal implementation of the AWED system [4, 5]. Note that while we present a Java-based implementation (and an evaluation of Java-based middlewares in the following section), conceptually our approach is not tied to Java. The Arachne aspect system, for instance, features (non causal) regular sequence pointcuts for C applications and has been applied to the modification of network protocols used for the communication in distributed systems [11].

In the following, we first present the overall architecture of the resulting system. Second, we discuss how AWED can be used to test causality on distributed infrastructures that have not been prepared for the provision or use of causality information. Third, we discuss the implementation of the framework that supports causal finite state machines to support causal sequences without message reordering. Finally, we will present the mechanisms for message reordering that were included to support the pointcut construct `seqCausalOrder`.

4.1 AWED Architecture

AWED is a dynamic aspect language that weaves aspects with classes at load time and allows aspect deployment and undeployment at execution time. Its implementation presents an optimized partially evaluated interpreter for distributed aspects. Figure 4 shows the overall architecture, *i.e.*, its compilation chain and the main structures of its runtime framework. In the top left part of the figure we can see how the application and aspect code is compiled into Java bytecode. The bytecode is then read by AWED's instrumentation and transformation framework at load time, producing a version of the application that is instrumented at the necessary joinpoints (here a subset of the method calls). When executing the instrumented application, and once it reaches an instrumented joinpoint, the application dispatches joinpoint notifications to the `Registry` framework that takes care of the recognition of distributed sequence pointcuts. This framework passes the joinpoint notification to each aspect instance, that, in turn, evaluates each joinpoint to match pointcuts and to apply advice. An AWED runtime framework, including a registry, is running at runtime on each logical host, *i.e.*, JVM. In order to support remote pointcuts each registry, *i.e.*, each JVM, communicates joinpoint notifications to the other JVMs using an extension of the JGroups framework [16], one of the most popular Java-based middleware for group communication. This part of the infrastructure contains all necessary support for non-causal event relationships, in particular remote regular sequence pointcuts.

In figure 4, we have also detailed the two main extensions incorporated to the runtime framework in order to support the causal constructs. First, the communication framework (see the box labelled "JGroups extension" in the figure) has been extended to support causality-supporting protocols. The extended JGroups component uses the original JGroups framework augmented with specific protocols for causality. In the figure we show a traditional protocol stack that supports different protocols, including the User Datagram Protocol (UDP). The protocol stack shows, at the top, the `Causal AWED` protocol. This protocol can be any of two new protocols that we have implemented. Second, the pointcut class hierarchy (see the class diagram for causal pointcuts highlighted in magnifying glass in the figure) has been augmented by support for causal sequence-based aspects, concretely by support for causal pointcuts with or without reordering and a notion of transition guards. In the following we present both extensions in some more detail.

Causality-supporting protocols. The two new protocols that support causality do not modify actual communication, but just handle causality and delegate actual communication to the other protocols in the protocol stack. The first protocol that we have implemented is the `Causal tags + clock increase` protocol. This protocol tags the distributed messages with a vector clock time, and will calculate the value of the new vector clock times at a host upon arrival of new messages. This protocol can be used to detect causal relations, but it can not

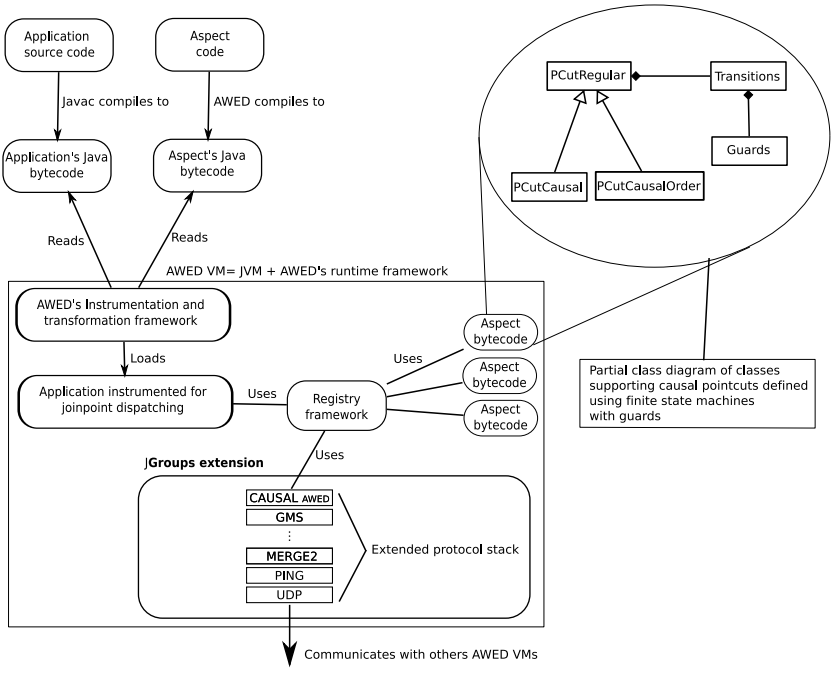

Fig. 4. AWED architecture

be used to impose causal ordering of messages. The second protocol that we have implemented is `Causal tags`. This is a more lightweight protocol that tags messages with vector clock times but does not update the vector clock. This protocol can be used with specialized adapters to add causality information to distributed infrastructures and applications that have not been aware of causality information in the first place.

4.2 Adding Causality to Non-causal Distributed Applications

Most distributed infrastructures and applications do not implement causality natively. Adapting such applications to support causality typically is very cumbersome and error prone. To avoid this problem, we propose specialized adapters that can be used to instrument causality transparently in legacy applications. To prove that this is a feasible solution we have implemented an adapter for RMI based applications, thus covering a wide spectrum of distributed Java applications. This adapter is realized using Java's notion of customized sockets.

The adapter basically implements a mechanism similar to that provided by the `Causal tags + clock increase` protocol. Thus, each message in the legacy application is now tagged with a vector clock and a local vector clock is updated upon arrival of each RMI message. This connector can be combined with the AWED framework that is running the `Causal tags` protocol to detect causal relations in the legacy application. This deployment do not need any particular modification of the legacy application. To use the specific connector, the programmer just specifies an option for the JVM when invoking AWED.

Causal sequence constructs with guarded finite state machines. In order to implement the causal sequence construct as presented in section 3 we have modified the compiler and the runtime infrastructure of the previous non-causal execution system of AWED. The previous AWED system has already used finite state machines to support regular sequence pointcuts. The corresponding implementation evaluates each join point and, depending on the current state of the automaton, accepts or rejects a joinpoint. In case of acceptance, a state transition is executed before executing the advice. We have extended this model to support guards. Thus, at compile time the state machine is constructed with specific guards, mainly to support the causal tests required by causal relationships expressed using the `conc` or `causal` transition modifiers.

At runtime, the new execution system includes two major extensions. First, before accepting or rejecting a joinpoint, the state machine evaluates the corresponding guard, *e.g.*, the causal information of the current joinpoint, and if the guard is satisfied the joinpoint is evaluated. The second modification address the management of vector clocks: evaluation of causal regular sequences has to compute a new value for the vector clock each time that it accepts a joinpoint. This approach has a major benefit compared with other frameworks implementing causality: finer grained control over events tagged with vector clocks and, as a consequence, less performance overhead.

To implement the causal sequence construct with reordering we have further extended the automata-based pointcut recognition component. Each such

component now has its own vector clock that is advanced each time a message is processed (including messages not in the alphabet of the state machine). To address reordering, the state machine uses a delay queue where it stores the messages that do not arrive in the right (causal) order. The messages in this queue are causally ordered but not necessarily consecutive. Upon arrival of a new message it gets evaluated: if it is accepted and if the message causally is the next message with respect to the vector clock of the state machine, it is processed and the first message in the delay queue is evaluated again.

Finally, a note on the scalability of our approach: Concerning scalability of the pointcut matching, the principal property is that the AWED architecture (cf. Fig. 4) does not impose any centralized control, in particular, for the monitoring of pointcuts that involve causal relationships. The other components of the AWED architecture (principally matching of other pointcut types and execution of remote advice) do not require central control either as discussed as part of our previous work [4].

5 Evaluation

In this section we present a qualitative and quantitative evaluation of our approach using JBoss Cache [15], a Java-based middleware infrastructure for replicated caching (part of JBoss middleware tools). First, we analyze a non-trivial test case for replicated caching and show that aspects based on control-flow and causal patterns significantly improve the corresponding debugging and unit testing tasks. Second, we evaluate the performance of our prototype implementation in a two-fold manner. A series of micro-benchmarks provides evidence that our implementation supports regular causal sequences with no to reasonable small performance overhead. Finally, in order to provide concrete evidence that we meet the objectives set out in the motivation, we compare AWED's use of sophisticated regular causal sequences to the use of the Eclipse debugger as a popular tool for the debugging of distributed Java applications by means of loose coordination of per-host debugging sessions.

5.1 Qualitative Evaluation

In the following we present a qualitative evaluation of our approach involving debugging and testing scenarios in two Java-based middlewares, JBoss Cache [15] and Apache's ActiveMQ [27].

Deadlock testing in JBoss Cache. In JBoss Cache (Ver. 2.0.0GA) the method `performTest` of class `ReplicatedTransactionDeadlockTest` (see Fig. 5) implements a test case to detect a deadlock bug. The test case uses two caches, actions on the first cache are replicated onto the second cache by means of the replication framework. The method triggers multiple workers in multiple threads. Each worker starts a transaction, puts a value in the cache (all workers use the same memory position in the cache) and commits the transaction. The test has to be

```
1  private void performTest() throws Exception {
2    // repeat the test several times since it's not always reproducible
3    for (int i = 0; i < NUM_RUNS; i++) {
4      if (exception != null) { // terminate the test on the first failed worker
5        fail("Due to an exception: " + exception); }
6      // start several worker threads to work with the same FQN
7      Worker[] t = new Worker[NUM_WORKERS];
8      for (int j = 0; j < t.length; j++) {
9        t[j] = new Worker("worker " + i + ":" + j); t[j].start(); }
10     // wait for all workers to complete before repeating the test
11     for (Worker aT : t) aT.join(); } }
```

Fig. 5. Deadlock detection test case method

repeated a number of times (first `for` block in the figure) since it can't be reproduced easily. The original bug occurred 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. In this case, 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.

A programmer dealing with that bug faces tree problems: (i) how two reproduce the problem, (ii) how to debug it and (iii) how to write a suitable test case to identify it in the future. To deal with the first problem the code shown in Fig. 5 triggers several threads that execute transactions concurrently, hoping for the bug to be reproduced. This approach is subject to several problems, in particular, that a unit test session could pass over the bug without noticing it. Regarding the second problem, as part of a corresponding debugging session a programmer would have to apply a breakpoint either to the line for remote prepare or in the line that throws the corresponding exception. In the first case the debugger will stop on each prepare (buggy or not). In the second case it will, eventually, stop only on an error of one of the threads. Then, depending of how threads are scheduled, it could stop the application(s) in a buggy state or in a correct state, because the other action could have or have not enough time to complete the transaction. Additionally, the programmer could perform many runs without reproducing the bug. A test case for this bug is, of course, subject to all the problems detailed above.

Using our approach we can improve on the three development scenarios: debugging, unit testing and bug reproduction. Fig. 6 shows a pointcut that can be used to define a breakpoint that will occur only if the bug appears. The pointcut implements a sequence (*i.e.*, finite state machine) with three states and three transitions. The first state accepts a call to the method `runPreparePhase`, from the `ReplicationInterceptor` class in the cache that belongs to the `source` group (`source` and `target` are dynamic groups that can be handled using AWED). Once such a method is received, the state machine changes its state to a state that accepts `tCommit` transitions and `tSecondPrepare`, the latter representing a prepare operation issued by another worker. If the target cache receives

```
pointcut deadlock():
  s1:seqCausalOrder(
    tPrep:
      call(* ReplicationInterceptor.runPreparePhase(..)) && host(src) > tCommit || t2ndPrep,
    tCommit: call(* PessimisticLockInterceptor.commit(..)) && host(targ) > tPrep,
    tSecondPrepare: call(* ReplicationInterceptor.runPreparePhase(..)) && host(src)) &&
  step(s1, tSecondPrepare);
```

Fig. 6. Pointcut for deadlock detection in a synchronous transactional cache

```
pointcut prepare(): call(* ReplicationInterceptor.runPreparePhase(..)) && host(src);
pointcut commit(): ... && call(* BaseRpcInterceptor.replicateCall(..)) && ...

pointcut generateDeadlock():
  s1:seqOrderedCausal(
    tPrep : prepare() > tCommit || t2ndPrep,
    tCommit : commit() > tCommit || t2ndPrep,
    t2ndPrep: prepare() );

before(): generateDeadlock() && step(s1, tCommit) { while(block){ Thread.yield(); } }
after(): generateDeadlock() && step(s1, t2ndPrep) { block=false; }
```

Fig. 7. Aspect ensuring the generation of the buggy behavior for deadlock detection

a `tCommit` message, the normal behavior, it returns to the first state. Finally, if the sequence detects, after the first `tPrepare` message, a `tSecondPrepare` message on the `source` cache, the state machine recognizes a deadlock state. Note that the sequence definition must be ordered causally in order to ensure that the events will be detected in the correct order in any distributed setting.

AWED's regular causal pointcut definitions can also be helpful for bug reproduction and unit testing. The main problem with current test case definitions, such as that introduced above, is that it is of haphazard nature, *i.e.*, it does not always allow to reproduce the bug situation. Figure 7 shows an excerpt of code from an aspect that will interact with the original test case of Fig. 5 to impose the desired order of events in the presence of only two workers. The aspect excerpt includes the definition of a state machine that matches a call to the method `runPreparePhase`, which means that the corresponding transaction has acquired the lock and is going to broadcast a prepare message to the target cache. Then, if it detects a call to the `replicateCall` method having as parameter a commit method call, a before advice will suspend the current thread until another `runPreparePhase` is detected. A buggy implementation will allow this reordering of events, a correct implementation will produce a lock-timeout exception because the cache node will be locked by the second transaction.

Debugging ActiveMQ. We have also performed experiments over the Apache project's ActiveMQ message broker [27] that is used, *e.g.*, for the integration of enterprise information systems. From an analysis of the list of the 359 open issues in ActiveMQ's bug tracking system as of Aug. 2008, we have found six issues classified as *blockers*: at least four of these are caused by the wrong ordering of events or messages. Similarly, out of the 13 messages classified as *critical* at least

five are related to message or event ordering. We have successfully woven causal aspects on ActiveMQ. To test the applicability of our approach we have debugged a use case regarding a deadlock situation in a configuration setting with four brokers and a use case involving the wrong ordering of repeatedly delivered messages in the context of transactions session with roll back. In both cases we have successfully defined simple pointcut definitions that exactly test for the corresponding error situations. These tests provide evidence that our approach, in particular the AWED system, is applicable generally to Java-based middleware. Finally, as for JBoss Cache, these debugging experiments have incurred only minimal overhead in both the Java client and the ActiveMQ broker.

5.2 Micro-benchmarks

We have run performance tests of our implementation using the performance framework of JBoss Cache. This framework allows to run multiple performance test over cache configurations. The tests were performed in a cluster of 4 nodes. Each node was equipped with a double core AMD Opteron 250 (2400 MHz) processor in 32 bit mode, 4 GB of memory and a 1 GB network interface. The *test case scenario* we have used is the default Web-Session simulator of the JBoss Cache framework that basically simulates the interaction of a replicated http session in a cluster of application servers. This test can be parametrized on the number of requests and the ratio of reads to writes requests.

We have evaluated the performance of the extended protocols developed to support causality in AWED. To this end, we have compared four different protocol configurations: (i) the performance of JBoss Cache with a standard, non-causal, configuration of its communication protocol stack (denoted `Normal` below), (ii) the causality protocol `Causal` natively provided by JGroups and (iii) our new protocols `Causal tags` and `Causal tags + clock increment`.

Table 1 shows the results of several test sessions in our cluster. The first set of sessions was performed with a ratio of 80% reads and 20% writes over 100.000 operations (left part of the table) and the second set of tests considers a ratio of 20% reads and 80% writes (right part of the table). Each node in the test executes 100.000 requests and only the writes are replicated to the other members. The data shows that in both cases the `Normal` protocol and the `Causal tags` protocol presents the best performance average For the test with 20% writes, the `Causal` protocol (full causality, *i.e.*, vector clocks, clock

Table 1. Test results of 100.000 requests with respectively 20% and 80% writes

Protocol	Requests per second			
	20% writes		80% writes	
	Average	Standard dev.	Average	Standard dev.
Normal	63,350.23	7,004.93	58,033.77	9,792.51
Causal	60,961.14	11,867.69	53,814.05	7,085.89
Causal tags + clock inc.	52,107.34	27,790.92	53,463.53	7,310.65
Causal tags	60,396.03	7,420.05	59,487.43	7,405.64

increment and reordering) presents lower performance overhead than the `Causal tags + clock increment` protocol. Overall our new protocols do not impose a significant performance overhead (especially in the case of a large number of writes to the cache) compared to the standard JBoss Cache protocols.

5.3 Remote Debugging vs. Distributed Debugging

In order to provide evidence that we have achieved the main objective set out in the motivation part, that is, that regular causal sequences improve on a per-host approach to debugging, we compare the performance of a remote debugging session with Eclipse and a distributed debugging session with AWED. To this end, we have again used the JBoss Cache benchmark framework. We first compare two debugging sessions, one with Eclipse and one with AWED, without breakpoints in order to measure the overhead of the frameworks. We then compare both debugging sessions in the presence of a high-frequency breakpoint (*i.e.*, reached and fired many times).

AWED runtime overhead vs. Eclipse remote debugging overhead. Table 2 (left part) compares the overhead of the debugging infrastructure posed by eclipse in a debugging session and the overhead posed by our AWED prototype. This test doesn't include any breakpoint, thus it only compares the overhead of the execution frameworks. The table shows small and comparable overhead for both frameworks. This is not surprising due to the fact that both frameworks are based on the Java agent technology and no breakpoints are evaluated.

As a last experiment we have compared the overhead of Eclipse and AWED in the presence of a high-frequency breakpoint: a breakpoint in the method `invoke` of the interceptor class ReplicationInterceptor. Table 2 (right part) shows the behavior of the Eclipse debugger attached to four nodes running the JBoss Cache framework and the behavior of AWED breakpoints under such conditions. In table 2 the protocol configuration labeled as *invasive causality* implies that the application being debugged has been invasively modified with an adapter for causality, thus AWED system can predicate over application's own messages. Using the Eclipse debugger we have executed the benchmarks first in JBoss Cache normal configuration and then with JBoss Cache using JGroups default `CAUSAL` protocol. The performance in these configurations is very bad and after several problems with memory overflow and unacceptable delays for the test we have reduced the number of request to 100. On the other hand, the test of

Table 2. Debugging session without breakpoints (left half) and with a high-frequency breakpoint (right half)

	Protocol	Requests per second		No. of	Requests per second		No. of
		Average	Std. dev.	requests	Average	Std. dev.	requests
Eclipse Debugger	Normal	55,111.79	7,792.45	10^5	2.80	0.21	100
	Causal	55,172.60	5,764,97	10^5	3.39	0.30	100
AWED	Causal tags + clock inc.	56,079.85	5,983.75	10^5	234.77	5.07	10^5
	Invasive causality	53,045.19	10,223.90	10^5	237.61	7.58	10^5

performance using the AWED framework are at least seventy times faster and do not impose any restrictions in the conditions of the test. This is due to the fact that, even tough the Eclipse debugger and AWED's dynamic framework use similar execution technology, AWED implements several optimization techniques and was designed with distribution in mind [6]. Our approach thus scales much better than the discussed debugging methods using Eclipse.

6 Related Work

Our approach for causality is based on the idea of causality based on vector clocks introduced by Mattern [19] (that itself extended Lamport's approach on logical time introduced in the landmark pape [17]). These results were later integrated into actual middleware for reliable distributed systems based on group communication,*e.g.*, see the Horus framework [28]. The benefits and limitations of using causal communications, in particular, the resulting overhead that is added to all communication, has been actively discussed [7, 8, 24]. Our approach extends similar current approaches, *e.g.*, the support for causality in JGroups [3] We have provided concrete evidence that expression of causal communication at the language level is useful in the presence of real-world debugging scenarios in current middleware.

Debugging of control-flow based relationships between execution events has been one of the main domains of application of causality and logical clocks, see *e.g.*, [10, 13, 14, 23, 25]. Hseush et al. [14] and Ponamgi et al. [23] have presented *Data Path Expressions* (DPE), a control-flow based debugging language for concurrent applications. Our sequence construct combined with the pointcut language provide similar flexibility as their theoretical language, additionally we provide a fully distributed solution with no central monitoring component.

More recently Sen et al. [25] proposed an algorithm for decentralized monitoring used to check violations of safety properties in distributed systems. Monitoring expressions in their approach are written in past time linear logic. Their proposal presents *knowledge vectors* (inspired by vector clocks) and propose the Diana tool and actors as an implementation support. Our approach provides richer expressivity because of our general notion of transition guards and allows group relationships to be expressed.

Other approaches have addressed the implementation and formalization of distributed models for debugging (*e.g.*, see [10, 21]). However, either they do not consider the causality concept and ordering of events (*e.g.*, De Rosa *et al.* [10]) or, they restrict the concept of causality to the concept of distributed control flow (*e.g.*, Mega and Kon [21] as well as Li's work on monitoring of component-based systems [18]) These approaches can only express a small subset of the relationships we consider. Finally, control flow relationships for the debugging using aspects have been considered only for the sequential case, notably by Chern and De Volder [9].

7 Conclusion

In this paper, we have argued for the use of programming abstractions as expressive support for the debugging and testing of distributed middleware, in particular for the definition of sophisticated relationships between distributed events and the recognition of event sequences in the presence of non-deterministic executions. We have presented a corresponding aspect-based language and implementation support that introduces causal event sequences into AWED, an aspect system for the dynamic manipulation of distributed systems. We have validated our approach in the context of Java-based middleware, in particular for the debugging and unit testing of a JBoss Cache and Apache's ActiveMQ. This evaluation has shown that our implementation has reasonable overhead and that our approach significantly improves on the use of debuggers, such as Eclipse, that are based on the manual coordination of per-host debugging sessions.

This work paves the way for several leads of future work. On a conceptual level, more flexible abstractions to define relationships that mix events that partially are causally ordered and partially are not are of foremost interest. Furthermore, exploring the use of our abstractions in other application domains, such as grid infrastructures, should be explored.

References

1. Anderson, J.H.: Lamport on mutual exclusion: 27 years of planting seeds. In: PODC 2001: Proceedings of the twentieth annual ACM symposium on Principles of distributed computing, pp. 3–12. ACM Press, New York (2001)
2. Awed home page (2008), `http://www.emn.fr/x-info/awed`
3. Ban, B.: JGroups, reliable multicast comm. (2002), `http://www.jgroups.org/`
4. Benavides Navarro, L.D., Südholt, M., et al.: Explicitly distributed AOP using AWED. In: Proceedings of the 5th ACM Int. Conf. on Aspect-Oriented Software Development (AOSD 2006). ACM Press, New York (2006)
5. Benavides Navarro, L.D., Südholt, M., Vanderperren, W., De Fraine, B., Suvée, D.: Explicitly distributed AOP using AWED. Research Report 5882, INRIA (March 2006)
6. Benavides Navarro, L.D., Südholt, M., Vanderperren, W., Verheecke, B.: Modularization of Distributed Web Services Using Aspects with Explicit Distribution (AWED). In: Meersman, R., Tari, Z. (eds.) OTM 2006. LNCS, vol. 4276, pp. 1449–1466. Springer, Heidelberg (2006)
7. Birman, K.: A response to cheriton and skeen's criticism of causal and totally ordered communication. SIGOPS Oper. Syst. Rev. 28(1), 11–21 (1994)
8. Cheriton, D.R., Skeen, D.: Understanding the limitations of causally and totally ordered communication. In: SOSP, pp. 44–57 (1993)
9. Chern, R., De Volder, K.: Debugging with control-flow breakpoints. In: AOSD 2007: Proceedings of the 6th international conference on Aspect-oriented software development, pp. 96–106. ACM, New York (2007)
10. De Rosa, M., Goldstein, S.C., Lee, P., Campbell, J.D., Pillai, P., Mowry, T.C.: Distributed watchpoints: Debugging large multi-robot systems. International Journal of Robotics Research (2007)

11. Douence, R., Fritz, T., Loriant, N., Menaud, J.-M., Ségura-Devillechaise, M., Südholt, M.: An expressive aspect language for system applications with arachne. In: Proc. of AOSD 2005. ACM Press, New York (2005)
12. Eclipse Foundation. Remote debugging in Eclipse (2008), http://www.eclipse.org
13. Fowler, J., Zwaenepoel, W.: Causal distributed breakpoints. In: Proceedings of the 10th International Conference on Distributed Computing Systems (ICDCS), Washington, DC, pp. 134–141. IEEE, Los Alamitos (1990)
14. Hseush, W., Kaiser, G.E.: Modeling concurrency in parallel debugging. In: PPOPP, pp. 11–20 (1990)
15. JBoss Cache home page (2008), http://labs.jboss.com/jbosscache
16. JGroups home page (2008), http://www.jgroups.org
17. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21(7), 558–565 (1978)
18. Li, J.: Monitoring and characterization of component-based systems with global causality capture. In: 23th Int. Conf. on Distributed Computing Systems, Providence, RI. IEEE, Los Alamitos (2003)
19. 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 (October 1988)
20. Mega, G., Kon, F.: Debugging distributed object applications with the Eclipse platform. In: Eclipse 2004: Proceedings of the 2004 OOPSLA workshop on eclipse technology exchange, pp. 42–46. ACM, New York (2004)
21. Mega, G., Kon, F.: An Eclipse-Based Tool for Symbolic Debugging of Distributed Object Systems. In: Meersman, R., Tari, Z. (eds.) OTM 2007, Part I. LNCS, vol. 4803, pp. 648–666. Springer, Heidelberg (2007)
22. Nishizawa, M., Shiba, S., Tatsubori, M.: Remote pointcut - a language construct for distributed AOP. In: Proc. of AOSD 2004. ACM Press, New York (2004)
23. Ponamgi, M.K., Hseush, W., Kaiser, G.E.: Debugging multithreaded programs with MPD. IEEE Software 6(3), 37–43 (1991)
24. Schwarz, R., Mattern, F.: Detecting causal relationships in distributed computations: in search of the holy grail. Distrib. Comput. 7(3), 149–174 (1994)
25. Sen, K., Vardhan, A., Agha, G., Rosu, G.: Efficient decentralized monitoring of safety in distributed systems. In: ICSE, pp. 418–427. IEEE, Los Alamitos (2004)
26. Allinea Software. Distributed debugging tool (2008), http://www.allinea.com/
27. The Apache software foundation. Apache ActiveMQ is an open source message broker (2008), http://activemq.apache.org/
28. van Renesse, R., Birman, K.P., Maffeis, S.: Horus: a flexible group communication system. Commun. ACM 39(4), 76–83 (1996)