Invasive patterns for distributed programs

Luis Daniel Benavides Navarro, Mario Südholt,
Rémi Douence and Jean-Marc Menaud

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

Abstract. Software patterns have evolved into a commonly used means to design and implement software systems. Programming patterns, architecture and design patterns have been quite successful in the context of sequential as well as (massively) parallel applications but much less so in the context of distributed applications over irregular communication topologies and heterogeneous synchronization requirements.
In this paper, we propose a solution for one of the main issues in this context: the need to complement distributed patterns with access to execution state on which it depends but that is frequently not directly available at the sites where the patterns are to be applied. To this end we introduce *invasive patterns* that couple well-known computation and communication patterns like pipelining and farming out computations with facilities to access non-local state. We present the following contributions: (i) a motivation for such invasive patterns in the context of a real-world application: the JBoss Cache framework for transactional replicated caching, (ii) a proposal of language support for such invasive patterns, (iii) a prototypical implementation of this pattern language using AWED, an aspect language for distributed programming, and (iv) an evaluation of our proposal for refactoring of JBoss Cache.

1 Introduction

Software patterns have proven a versatile tool for program development, be it for the development of application designs [15], architecture descriptions [24] or program implementations [14]. Design patterns have been very successful in the domain of sequential, in particular object-oriented applications. Similarly, pattern-based development methods have been extensively applied in the parallel domain for the derivation and implementation of massively parallel algorithms [24, 21, 9]. However, pattern-based approaches have been much less successful in the domain of distributed programming, in particular, if they are defined over irregular communication topologies and subject to heterogeneous synchronization constraints. Consequently, patterns for distributed programming (see, for instance, patterns

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

for distributed enterprise information systems and grid applications [20, 10, 14]) are often expressed as mere programming recipes that are not backed up by concrete architecture or implementation entities that can be used and reused as building blocks for applications.

In this paper we investigate a major reason for the difficulty in applying programming patterns, that embody common computation and communication patterns, to distributed applications: frequently, applications of such patterns in realistic contexts depend on information on the execution state that is not directly available when the pattern is to be applied. This is, for instance, the case in two frequent cases: (i) in legacy contexts where patterns could be used to improve the application structure but in which instructions for communication instructions and manipulation of related execution state are frequently scattered over numerous places and (ii) distributed applications that have been designed using less flexible abstractions than provided by communication and computation patterns.

In this paper we introduce *invasive patterns* for distributed programming. Such patterns essentially provide well-known regular computation and communication patterns but provide a built-in abstraction for access to non-local execution state whose access is required to enable pattern applications. We provide evidence that techniques from Aspect-Oriented Programming (AOP) [1] can be harnessed to augment patterns by structure access to such non-local state.

Concretely we present the following contributions. First, we present a detailed motivation for invasive patterns and corresponding aspect-oriented support based on a detailed analysis of the use of patterns in a real-world distributed application: the JBoss Cache strategy for replication in the context of transactions. Second, we introduce a pattern language that allows to concisely define invasive variants of well-known patterns for distributed applications. Third, we briefly sketch a prototypical implementation of invasive patterns using the AWED system for explicit distributed aspect-oriented programming. Fourth, we give an evaluation of our approach by discussion how invasive patterns can be used to improve the structure of JBoss Cache.

The paper is structured as follows. In Sec. 2, we introduce the notion of invasive patterns and motivate it in a real-world application. Our pattern language is introduced in Section 3. In Sec. 4, we describe our prototypical implementation of this pattern language using AWED. Sec. 5 presents the evaluation of our approach. Related work is discussed in Sec. 6. Finally, Sec. 7 gives a conclusion and discusses future work.

2 Motivation

In this section we first present modularization problems of pattern-like computations in JBoss Cache [17], a large-scale real-world framework for transactional replicated caching. We then introduce the notion of invasive patterns that we propose as a means to resolve such modularization problems.

2.1 Pattern-like structures in JBoss Cache

We have analyzed the occurrences of pattern-like computation structures and the dependencies of such pattern-like structures on the underlying execution state in JBoss Cache, an open source implementation of a replicated transactional cache over an J2EE-based communication infrastructure. In the following we briefly describe the JBoss Cache framework and the results of our analysis of software patterns that are used implicitly in this infrastructure.

JBoss Cache is a large object-oriented framework implemented in Java that consists of more than 50 thousand lines of code. Basically the JBoss Cache implementation consists of two main parts: (i) a main class `TreeCache` that represents the main data structure, a tree with a hash table on each leaf, that is replicated on each node (host) in the cache cluster and (ii) a set of filters that is used to implement the major part of the behavior of non-functional requirements, mainly transactions and data replication. An interception mechanism is used to transfer control between the classes implementing the data structure and the filters. Concretely, each call to the `TreeCache` API is first transformed into a method call object using a reflection mechanism. Once this object is created, it is passed to a chain of filters where each filter adds some behavior, *e.g.*, optimistic locking is added by the transaction filter. Eventually, the filtered method call is performed.

The current production version (1.4) of JBoss Cache conceptually uses an architecture that can be expressed nicely in terms of patterns using, *e.g.*, a pipeline pattern for transaction control and a farm pattern for replication actions (Here and in the rest of the paper we assume the cache to be configured for transactions with pessimistic locking and a two phase commit protocol). Figure 1 presents a high-level pattern-based view of the corresponding system structure of JBoss Cache. In the figure, a transaction is triggered by a specific method call represented by the first node in the pattern. Then successive calls to `get`, `remove` or `put` methods on the cache are executed and the information is stored for further replication. When a particular value is not present in the cache, the cache looks for the value in a group of selected neighboring nodes, its so-called buddies, illustrated by the three edges starting in the second node of the figure. Once the end of a transaction is reached, the originating cache engages a two phase commit protocol. In such a protocol the originating cache sends a prepare message with the transaction control information (edges numbered 1 in the right part of the figure), followed by answers from all buddies confirming agreement or non agreement (edges numbered 2). Finally, the originating cache sends a final commit or a rollback message depending on the answers it received (edges numbered 3). Note that in this interaction we can identify, at least, two well separated groups of hosts, one for the search of values at buddy nodes and the other for the replication behavior from a node to other nodes.

In previous work [6] we have analyzed the complexity of the code structure of the (non-pattern based) implementation of the JBoss Cache framework: we have shown that replication and transaction instructions are, in particular, widely scattered over the code base and tangled with one another in numerous

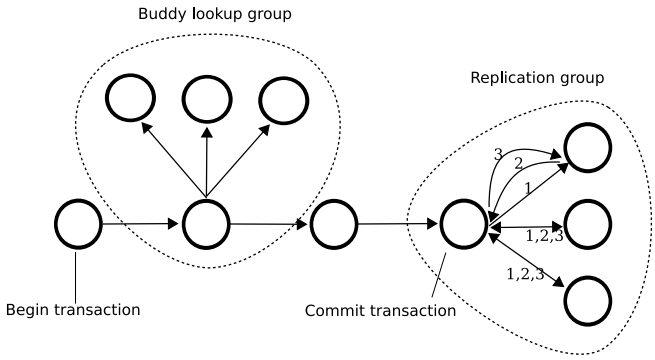

Fig. 1. Architecture of transaction handling with replication in JBoss Cache

places.* Even though JBoss Cache conceptually is characterized by a pattern-based structure as shown in Fig. 1, the current implementation does not allow conventional patterns for distributed systems to be applied due to the scattering and tangling of code these functionalities are subject to. The `TreeCache` class consists of 3802 lines of code (LOC), of which more than 280 LOC are relevant for transactions. The interceptor package exhibits similar quantitative characteristics: the package consists of 5099 LOC and more than 137 LOC are related to transactional behavior and are not included in the dedicated transaction interceptor. A detailed qualitative analysis of such code leads to the identification of three basic problems:

1. Transactional and replication behavior depends on state that is stored in different classes. Such state is modified in scattered pieces code that, *e.g.*, reify the current transactional state as mentioned above so that it can later be tested in another class in order to decide which replication action to perform.
2. The relationships governing the interplay between the main concerns, transactions and replication, are not made explicit anywhere in the code. Instead, scattered pieces of code implicitly coordinate these concerns, thus generating tangled code and breaking the modularization aimed at by the JBoss Cache filter mechanisms.
3. JBoss cache includes several distribution-related concerns (*e.g.*, replication, cache loaders and buddy lookup) that require communication between different groups of hosts. Group overlapping and interactions between different groups generate additional tangling.

2.2 Source code representation of pattern-like structures

These problems are clearly apparent in the source code of JBoss Cache. In abstract terms, a cache behaves as follows. A chain of interceptors for the support

* This analysis has been conducted on JBoss Cache version 1.2 but remains valid for the current production version 1.4 as shown in [23].

of transactional and replication behavior is created when the cache is initialized. When a transaction-related action occurs, a method-call object of a corresponding type is created using the JAVA reflection framework, which is then passed through the chain of interceptors. A "get" request, for example, may be processed by filter to check its buddies if a specific data is in their cache, by the transactions interceptor to control transactional behavior, and by the locking interceptor to lock the tree cache data structure accordingly. The so-called replication interceptor, finally, performs (most of) the two-phase commit protocol among caches by first sending a *prepare* message, followed by a *rollback* or *commit* message depending on the result of the prepare phase. This code architecture is problematic because the manipulation of state that is relevant for replication operations as well as the protocols governing transactional and replication behavior is determined by scattered pieces of code whose joint effects during execution, *i.e.*, the correct implementation of transactional behavior and replication, are very difficult to apprehend from the code structure.

Figure 2 shows a piece of code of the main filter method `invoke` of the `DataGravitationInterceptor` class that is responsible for the so-called data gravitation concern, *i.e.*, buddy lookup. This method clearly exhibits the problems stated above, providing evidence of tangling of three concerns: replication, transactions and buddy lookup. The code uses a common idiom to address transactions control inside a `switch` instruction (lines 7 to 21). The right branch in the switch statement is taken depending on static information on the execution state, *e.g.*, a configuration-time choice between optimistic and pessimistic locking, and dynamic information about the execution state, *e.g.*, the dynamic type of the current processed method call. There, in order to calculate the method id (see line 5) the application relies on an ad-hoc mapping that is defined in the class `MethodDeclarations`. Similarly, the choice between optimistic and pessimistic locking is made at configuration time inside the `TreeCache` class as well as part of the class `InterceptorChainFactory` (this choice in turn affects at runtime the configuration of the dynamically created chain of filters).

Note that the corresponding piece of code is found inside the filter class `DataGravitation` and uses data that is calculated in many different places, thus expliciting the problem 1 above. The kind of idiom involving `switch` statements (that clearly represent a mismatch between the conceptual pattern-based architecture and its concrete implementation) is scattered over multiple places in the implementation. We have found 93 places where such a switch action is used and more than 29 places where it occurs in the context of replication operations implying a *one to many* communication between caches (thus providing testimony for the problems 1 and 2 introduced above).

Furthermore, the `DataGravitation` class plays an unexpected role in the two phase commit protocol. A method of type `commitMethod` is processed in order to send a commit message on those caches that are not part of the current buddy group, see line 37 in the `docommit` method (*i.e.*, being subject to problem 3 above). Remember that the `DataGravitation` class was supposed not to con-

trol the transactional behavior or the replication of transactions which, should normally, be performed by the transactions and replication filters.

2.3 Invasive patterns in a nutshell

Dependencies as those motivated above for JBoss Cache between transaction-related actions and replication operations cannot simply be modularized using standard patterns for workflow-related computations, such as pipelining, farming out or gathering computations as illustrated in Fig. 3, where circles denote calculations that possibly take place on different hosts and edges denote communication. In fact, taking scattering and tangling of transactions and replication into account requires does not fit the common interpretation of such patterns in which each circle denotes a well-defined entity, in our motivating example some nicely modularized piece of code within JBoss Cache.

In such cases effective support for a pattern-based programming style should allow the definition of patterns to include accesses to the data it depends on but that is defined at other places in the underlying distributed program and allow such patterns to be applied possibly at numerous places in a program. Because crosscutting of non-local execution state that enables such pattern applications is at the heart of such effective support, Aspect-Oriented Programming [18, 1] seems a promising approach for the modularization of patterns and the corresponding data accesses.

We pursue this idea in this paper on the programming level by extending patterns with a notion of aspects to modularize such crosscutting accesses. The resulting notion of invasive patterns is illustrated in Fig. 4 for the case of a `gather` pattern. On the three nodes on the left hand side, different pointcuts (represented by dashed lines) are used to access information that is then prepared by "source" advice (represented by the filled rectangles) to be sent to the right hand side node. Once all relevant data has been passed to the right hand side node, a "target" advice is used to integrate the transmitted data with an existing or new computation on the target node. In order to support the declarative definition of such crosscutting accesses, we leverage results on so-called stateful pointcut languages [13] that enable matching of sequences of execution events to be defined using expressive languages, in particular finite-state automata.

Besides a definition of basic invasive patterns a suitable notion of pattern composition is needed. Reconsider the (abstract) architecture of transaction handling with replication in JBoss Cache, see Fig. 1: this architecture can naturally be expressed in terms of compositions of the three basic patterns introduced above, where the steps denoted 1–3 in the figure correspond, for instance, to two applications of the `farm` pattern and one application of the `gather` pattern. Our approach supports the compositional construction of such architectures from the basic patterns on the programming and the implementation level. As discussed in Sec. 2.1, this architecture is essentially hidden in the actual JBoss implementation. Our approach can therefore be seen as a means to make explicit such architectures, and thus help program understanding and maintainability.

```
1  //----- Piece of code in the invoke method of
2  //----- DataGravitationClass
3  try
4  {
5   switch (m.getMethodId())
6    {
7      case MethodDeclarations.prepareMethod_id:
8      case MethodDeclarations.optimisticPrepareMethod_id:
9           Object o = super.invoke(m);
10          doPrepare(getInvocationContext().getGlobalTransaction());
11          return o;
12     case MethodDeclarations.rollbackMethod_id:
13          transactionMods.remove(
14                 getInvocationContext().getGlobalTransaction());
15          return super.invoke(m);
16     case MethodDeclarations.commitMethod_id:
17          doCommit(getInvocationContext().getGlobalTransaction());
18          transactionMods.remove(
19                   getInvocationContext().getGlobalTransaction());
20          return super.invoke(m);
21   }
22 }
23 catch (Throwable throwable)
24 {
25  transactionMods.remove(
26                 getInvocationContext().getGlobalTransaction());
27  throw throwable;
28 }
29
30 //---- The docommit method in DataGravitation class
31 private void doCommit(GlobalTransaction gtx) throws Throwable
32    {
33       if (transactionMods.containsKey(gtx))
34       {
35          if (log.isTraceEnabled())
36               log.trace("Broadcasting commit for gtx " + gtx);
37          replicateCall(getMembersOutsideBuddyGroup(),
38             MethodCallFactory.create(
39                   MethodDeclarations.commitMethod,
40                   new Object[]{gtx}),
41                   syncCommunications);
42       }
43       else
44       {
45          if (log.isTraceEnabled())
46              log.trace(
47               "Nothing to broadcast in commit phase for gtx " + gtx);
48       }
49    }
```

Fig. 2. Tangled code of a two phase commit (2PC) protocol inside the invoke method of the `DataGravitationInterceptor` class.

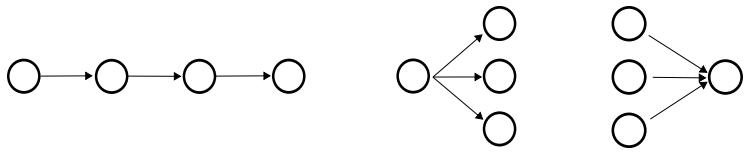

Fig. 3. Basic patterns

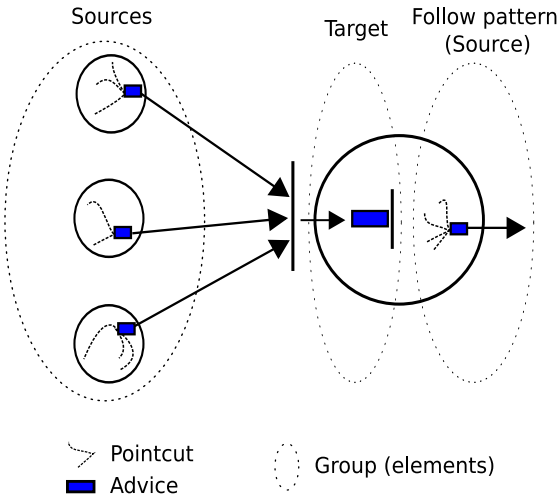

Fig. 4. Invasive patterns

3 Pattern language

A crucial issue concerning invasive patterns as motivated before is how the different activities (pointcut matching, local and remote advice) are synchronized with one another. In this section, we first discuss corresponding design choices and then present our language for the definition of invasive architectural patterns.

3.1 Design choices

The definition of distributed algorithms using patterns over a state-based programming paradigm essentially depends on the correct synchronization on the different parts of invasive patterns and between different invasive patterns. Pattern-based computations can be synchronized roughly at three different levels:

1. *Synchronization within an invasive pattern.* Most basically, target advice is executed only after a rendez-vous synchronization of all source computations. In the case of the `gather`-pattern shown in Fig. 4, the target computation is started only after the three target hosts have "agreed" to trigger it.
   Second, target advice may be executed in a synchronous or asynchronous fashion. Synchronous execution of parts of the `pipe` pattern of Fig. 3a corresponds to a fully sequential (a.k.a. batch) computation, while its asynchronous execution corresponds to a pipelined computation. We support both behaviors.
2. Computations involving *consecutive executions of patterns* may be synchronized with one another. The `gather` pattern may, for instance, be synchronized with the execution of the following pattern that is represented in the right hand side node by the pointcut (the dotted lines), the source advice

(the small rectangle) and the arrow leaving the node to the right. Execution of a follow pattern on a node n must obviously start after control of the previous pattern has entered n (otherwise the two pattern executions could not be said to be consecutive) but may be reasonably started either when the target advice of the previous pattern is started or when it terminates. In this paper we only consider the synchronous case, *i.e.*, execution of follow patterns start when the target advice finishes. Our prototype implementation already supports both options, though.

3. Most generally, synchronization constraints may be imposed on *arbitrary segments of pattern compositions*. Such general constraints are interesting, *e.g.*, because computations may be executed on the same host and therefore give rise to problems, such as race conditions. Such synchronization strategies cannot, however, be defined simply in terms of individual patterns as considered here.

Summarizing, we provide in this paper explicit support for intra-pattern synchronization and synchronization between consecutive pattern executions. We do not, however, provide general synchronization strategies over pattern compositions because they are difficult to comprehend and may easily lead to performance bottlenecks or even deadlocks. We envision that specific properties over pattern compositions can be analyzed and enforced in terms of the more restricted means for synchronization we introduce here. This issue is, however, beyond the scope of the present paper.

3.2 Syntax and informal semantics

$$
\begin{array}{lll}
P & ::= & \texttt{patternSeq}\ G_1\ A_1\ G_2\ A_2\ \ldots\ G_n \\
G & ::= & H\ G \ \mid\ P\ G \ \mid\ \epsilon \\
A & ::= & \texttt{aspect}\ \{\ \texttt{around}((H, \text{Id}^*)^*)\text{: } PCD \quad SourceAdvice \quad [\texttt{sync}]\ TargetAdvice\ \} \\
PCD & ::= & \texttt{call}(MSig) \ \mid\ \texttt{target}(Id) \ \mid\ \texttt{args}(Id+) \\
 & & \mid\ PCD\ \&\&\ PCD \ \mid\ PCD\ \|\|\ PCD \ \mid\ !PCD \\
 & & \mid\ Seq
\end{array}
$$

Fig. 5. Pattern language

We are now ready to introduce the pattern language we have designed that realizes the above design choices. Figure 5 shows the syntax of our pattern language (we have omitted details for the sake of simplicity).

The pattern constructor `patternSeq` takes as argument a list $G_1\ A_1\ G_2\ A_2\ \ldots\ G_n$ of alternating group and aspect definitions. Each triple $G_i\ A_i\ G_{i+1}$ in this list corresponds to a pattern application that uses the aspect A_i to trigger the pattern in a source group G_i and realize effects in the set of target hosts G_{i+1}. A group G is either defined as a set of host identifiers H or through a pattern constructor term itself. In the latter case, the group is defined as

the source or target group of the constructor term depending on the argument position the term is used in. This constructor enables to define the basic patterns shown in Fig. 3: `pipe` as a `patternSeq` from a single host to another, `farm` as a `patternSeq` from a single host to several hosts and `gather` as a `patternSeq` from several hosts to a single one. Pattern compositions can be defined with more complex `patternSeq` terms. For instance, the left hand side of Figure 6 defines a composition `pipe` then `farm`, and its right hand side defines a composition `pipe`, `farm` then `gather`. These examples make clear it is easy to define sophisticated compositions akin to the architecture of transaction handling in JBoss Cache (cf. Fig. 1).

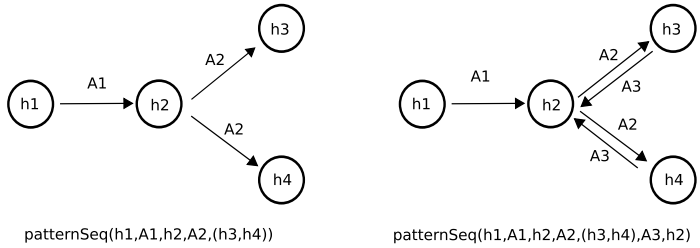

Fig. 6. Pattern Compositions

Aspects A that define the behavior of invasive patterns specify a pointcut *PCD* that allows to modularize crosscutting code that triggers a pattern, and define a source advice and a target advice executed respectively on the source and target groups of a pattern. Advice can be parametrized by source hosts H and bound values (see `args` below). An advice is a standard block for code, but a source advice can call the matched base call with the `proceed` keyword. Otherwise, the base call triggers the aspect but the execution of the corresponding base method is skipped. When a `sync` annotation is used to qualify target advice, the base program execution on source hosts is not resumed before the end of the target advice. The default behavior is asynchronous execution.

We consider pointcut definitions that, for presentation purposes, are essentially restricted to matching of method call joinpoints, may extract target objects with `target` and arguments of calls with `args` and use logical compositions of pointcuts. Following the paradigm of stateful pointcuts [13, 6] (and unlike AspectJ [4, 19]) pointcuts may match sequences (non-terminal *Seq*) of calls in the base program execution. We omit the syntax of sequences for now, but they are basically defined in terms of a finite-state automaton by declaring its states and by labelling state transitions with pointcuts.

Let us consider a small example. The aspect in Figure 7 profiles session creation. When the method `login` is called the local advice performs it (through a call to `proceed()`) and the target advice increments the integer counter defined within the aspect. This aspect can be applied using `patternSeq` to two hosts so that sessions on the first host are counted on the second.

```
1 aspect Profiling {
2   int sessions=0;
3   around(): call(* *.login()) { proceed(); } { sessions++; }
4 }
```

Fig. 7. A Session Profiling Aspect

4 Implementation

In order to implement the pattern language presented in the previous section, support for three main mechanisms is necessary: (i) aspects providing a modular abstraction for invasive access on the source hosts and triggering activities on target hosts, (ii) flexible means for synchronization within individual patterns and between consecutive pattern executions, and (iii) the concise definition of the communication topologies of patterns.

Mainstream sequential AOP languages, in particular AspectJ [19], do not fit well these requirements because they do not include any specific support for distribution and concurrency and are therefore subject to well-known deficiencies if used for the modularization of distribution concerns (as exposed, *e.g.*, by Soares et al. [22]). Concretely, with regard to invasive patterns such aspect languages would require to split the definition of patterns into different, at the application-level unrelated aspects that have to be manually deployed on different hosts.

We have implemented invasive patterns using a recent approach to AOP for distributed applications, Aspects with Explicit Distribution (AWED) [6, 5], which provides direct support for most of the necessary features and allows to accommodate the remaining ones based on its native abstractions. The AWED language has been designed as an aspect language for the modularization of crosscutting concerns in distributed systems. In general terms, AWED allows to define pointcuts that match sequences of execution events on different hosts in a distributed systems that trigger advice that is executed on potentially different hosts.

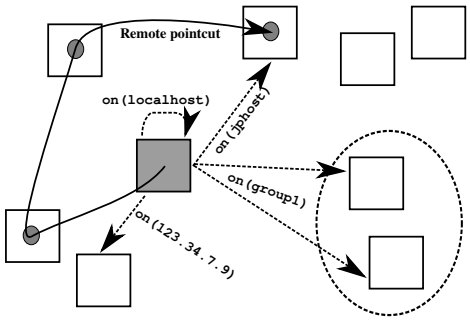

Fig. 8. Remote pointcuts and advice in AWED

Figure 8 illustrates the two main features of the language: remote pointcuts and advice. Pointcuts essentially allow to match sequences of execution events that occur on different hosts. Hosts can be referred to using absolute addresses but can also be defined relative to the host on which an aspect is deployed (term `localhost`, in the figure the host colored in gray). Remote advice can be triggered on other hosts using the `on` specifier. Besides the host specifications available for pointcut definitions, advice execution can also be specified to take place on the host where the pointcut has been matched (term `jphost`). Pointcuts and remote advice execution may depend on explicitly defined groups of hosts. In pointcuts, such groups may limit matching of execution events to sets of hosts; as to advice executions, groups allow to execute advice on several hosts. Furthermore, AWED allows to execute pieces of advice synchronously or asynchronously with the execution of the base application and with other aspects.

A farm pattern can be mapped to an AWED aspect having a pointcut expression as

`call`(* *.login()) && `host`("sources") && `on`("targets"),

there, the `call` pointcut matches calls to *login* method. The pointcut `host`("sources") matches the join points (events) that appear in a host that belongs to the *sources* group. Finally the pointcut `on`("targets") triggers the execution of the advice in hosts that belong to the *targets* group. AWED also supports the `Seq` pointcut that allows to specify finite-state automata that permit to match sequences of join points in distributed applications. The sequence constructor is used to map direct uses of *Seq* pointcuts of our pattern language and to implement *rendez-vous* synchronization in gather-like patterns. We have developed a formally-defined transformation from our aspect language into executable AWED programs.** More information on the concrete translation of programs expressed using the pattern language into AWED programs can be found along with substantial examples in the following evaluation section.

5 Evaluation

In this section we evaluate our approach by presenting how invasive patterns can be used to restructure transaction handling and replication in JBoss Cache We first how to implement these concerns using the proposed pattern language, thus making explicit their pattern-based structure. We then briefly discuss the resulting implementation in AWED. Third, we qualitatively evaluate the resulting pattern-based implementation by discussion the difference in conciseness of the original and new implementation. Finally, we briefly discuss first benchmarking results we have performed by executing the refactored implementation of JBoss Cache using the current AWED implementation [5].

** Note to reviewers: this formal transformation, that cannot be described here because of lack of space, is available on request.

5.1 JBoss Cache revisited

Invasive patterns allow to concisely express the essentials of the pattern-based architecture for transaction handling and replication in JBoss Cache as shown in Fig. 1. Concretely, we have implemented support for transactions with pessimistic locking and the two phase commit protocol using invasive patterns.

```
1   gCaches = {H1, H2, H3}
2   pipe([h],
3        Atransac,
4        farm(
5          gather(
6            farm([h], Aprepare, sync gCaches-[h]),
7            Apresp,
8            [h]),
9          Acommit,
10         gCaches-[h])
11       );
```

Fig. 9. Pattern-based definition of the JBoss Cache two phase commit

The corresponding solution is formulated in terms of a nested composition involving four pattern expressions, see Fig. 9. First, we apply a `pipe` pattern to be able to relate the start of transactions with the replication operations, *i.e.*, the start node and the final replication group, respectively, of Fig. 1. Once a commit is encountered, a `farm` pattern is used to farm-out the prepare phase of the two phase commit protocol. Then, a `gather` pattern is used to collect the answers from the involved buddy caches. Finally, after all answers have been received we use again a `farm` pattern to distribute the final decision of commit or rollback. The code in the figure defines this algorithm for three replicated caches. Note that replication can be triggered from any of the three caches. Once the triggering node (`h` in the algorithm) is selected the expression `gcaches-h` represents the group of caches without the triggering one.

Figure 10 shows the pattern-defining aspect `Aprepare` that farms out the prepare information of the two phase commit protocol. Occurences of calls to the `prepare` method are matched and executed (because of the call to `proceed` in the source advice). On the target hosts, the target advice executes the prepare phase followed by the invocation of an agreement or disagreement method, depending of the answer of the target caches. The aspect takes care of transactions that perform nested calls in the prepare method using the `cflow` pointcut construct: this constructs forbids new replication actions within the dynamic extent of an open call to the `prepare` method.

Implementation using AWED. The result of the transformation[***] of the pattern program shown in Fig. 9 is a set of AWED aspects that implement the

[***] we have applied the transformation manually for this evaluation but its automation is unproblematic.

```
aspect Aprepare {
  org.jboss.cache.TreeCache tc = CacheRegistry.getInstance().getCache();

  around(DataStorage d, String txId):
    call(* PrepareHelper.send(..)) && args(d,s) &&
    !cflow(call(TransactionManager.prepare(..)))

    // Source advice
    { proceed(); }

    // Target advice
    { TransactionManager tm = TransactionManager.getInstance();
      PrepareHelper ph = new PrepareHelper();
      try{
          tm.prepare(d, txId, tc);
        ph.respAgree(txId);
      } catch(Exception e) {
        ph.respNotAgree(txId);
      }
    }
}
```

Fig. 10. 2PC invasive aspect `Aprepare`

```
all aspect Aprepare_AWED {
  org.jboss.cache.TreeCache tc = CacheRegistry.getInstance().getCache();

  Group[] targetGs = {new Group("h1"), new Group("h2"), new Group("h3")};

  pointcut sourcePrepareCall(TransactionData d, String txId):
       seq(init:call(* Atransac.triggerNext()),
       pcd: call(* PrepareHelper.send(..)));

  pointcut targetPrepareCall(Transaction tx)(TransactionData d, String txId):
              call(* PrepareHelper.send(..));

  // source advice
  around(TransactionData d, String txId): sourcePrepareCall(d, txId) && host(localhost) {
    proceed();
  }

  // target advice
  after(TransactionData d, String txId): targetPrepareCall(tx) && on(targetGs) {
    TransactionManager tm = TransactionManager.getInstance();
    PrepareHelper ph = new PrepareHelper();
    try{
      tm.prepare(Tx.getTransacData(), Tx.getId(), tc);
      ph.respAgree(txId);
    } catch(Exception e) { ph.respNotAgree(txId); }
    void triggerNext() {};
}
```

Fig. 11. 2PC invasive AWED aspect for the creation of the transactional behavior

replication under pessimistic locking. Each aspect of the pattern-based solution is translated into an AWED aspect that modularizes source and target parts of a pattern expression. Figure 11 presents the resulting implementation of the `Aprepare` pattern-level aspect. In this case the generated source pointcut uses a sequence to explicitly relate the relevant transaction-related event to the call `send` that initiates replication, *i.e.*, farming out of the prepare action. The target advice executes the prepare method in the target caches and calls an `respAgree` or `respNotAgree` method to yield the answer.

5.2 Qualitative and quantitative evaluation

In Section 2 we have motivated that the current implementation of JBoss Cache is subject to problems concerning modularization, in particular, scattered and tangled code for the control of the transaction and replication concerns. Our solution improves the implementation in all those respects. First, each crosscutting concern is now modeled as an aspect and the choreography and interaction is defined without crosscutting by means of the pattern language (and AWED aspects on the implementation level). Second, distribution issues, coordination and composition of patterns are easily identifiable and modifiable in our solution. These advantages appear clearly in the `Aprepare` aspect: the source pointcut clearly defines the exact context (the sequence of method calls matched in the source pointcut) required to trigger the replication; furthermore, the related actions relevant to replication on different hosts are modularized in the aspect. Overall, our solution facilitates understanding and is easier to extend.

We have measured how our refactored version of JBoss Cache compares quantitatively to the plain JBoss Cache solution. For the corresponding experiments, we have considered transactions with pessimistic locking in JBoss cache. In the original code, there are more than 2674 LOC in 17 classes related to this concern. In our solution, the code consists of 532 LOC in 11 well-modularized aspects and classes: roughly a reduction of 80% of complexity (in terms of LOC). Most to this reduction is due to the fact that the transaction and communication protocol that is scattered and duplicated in `switch` structures is now re-factorized in well modularized entities.

6 Related work

As to the best of our knowledge there is no directly related work that considers extensions to standard communication and computation patterns to accommodate crosscutting data accesses using AOP techniques. However, there are many approaches that are related in a weaker sense, in particular, approaches that use AOP for support for pattern implementations, sequential AOP systems that have been used with distributed infrastructures and more generally pattern-based approaches in distributed systems. We consider these groups of approaches in the following.

There have been several recent articles on support for the implementation of patterns using AOP. Hannemann and Kiczales in [16], in particular, show that several quality attributes, such as locality of definition and code reusability, of GoF pattern implementations can be improved through usage of AspectJ. Technically, these improvements are achieved by representing some roles in the pattern more concisely using AO abstractions. This is quite a different endeavor from ours that focuses on AOP as a support technology for the definition of an extended notion of patterns. However, the results on sequential pattern implementations using AOP should have analogues for distributed patterns and should be applicable to some extent to the invasive patterns we advocate.

A number of approaches have been put forward that use sequential AOP systems like AspectJ for the modularization of crosscutting functionalities in distributed and concurrent applications. These approaches — such as Eric Tanter's work on ReflexD [25], recent work on implementations of concurrency operators [11] and the approach of Concurrent Event-Based AOP [12] — while in principle be able to express invasive patterns as we have proposed, can only do so by modularizing crosscutting functionalities using separate aspects for each node in a distributed system. Our approach, through its pattern language but also on the implementation level through transformation into AWED, is much more declarative by directly expressing distribution-relevant relationships within single aspects, thus resulting in more concise programs that facilitate program understanding.

In the domain of distributed applications, several pattern catalogues have been proposed [7, 20, 2]. Such patterns are particularly widespread in component-based systems, *e.g.*, the CORBA and J2EE platforms [8, 2]. These component systems provide communication and concurrency mechanisms that are used to implement patterns, *e.g.*, for the implementation of asynchronous broadcast services. However, these programming abstractions are not made explicit in the architectural description that defines the interconnection properties and, in contrast to our approach, no explicit means for the embedding of pattern-like interconnection structures in crosscutting contexts is provided.

In the more specific domain of (massively) parallel applications architectural and programming patterns are also quite popular. Much work has been done, for instance, on so-called skeletons following Cole's seminal work [9]. Recent work has focused on the application of such pattern-based parallelism to larger-scale imperative applications (see, *e.g.*, [24, 21]). Most of these approaches essentially rely on an underlying regular communication topology and use of a homogeneous synchronization model, two properties that do not hold for the applications we are targeting. Furthermore, crosscutting accesses to execution state on which pattern applications are not addressed explicitly in such approaches.

Finally, several authors have proposed configurable frameworks to address the implementation of complex communication protocols by composition of simpler protocol entities (see, *e.g.*, [26]). However such approaches address protocol composition at a much lower level of abstraction (*e.g.*, TCP, UDP connections) than we consider.

7 Conclusion and future work

Software patterns have proven a versatile tool for program development. They facilitate application development and maintenance by raising the abstraction level of descriptions for software artifacts. Patterns have been very successful for sequential object-oriented applications, as well as for massively parallel algorithms. However, pattern-based approaches have been much less successful in the domain of distributed programming that are defined on irregular topologies and subject to inhomogeneous synchronization requirements. In this papers we have identified a major reason for the difficulty in applying programming patterns to distributed applications: applications of such patterns frequently depend on information that is not locally available where the pattern is to be applied.

In this paper we have proposed a solution: *invasive patterns*. Such patterns provide well-known computation and communication patterns (e.g., pipe, farm and gather) but also offer a built-in abstraction based on AOP for access to non-local state. We have motivated our approach in the context of JBoss Cache, a real-world infrastructure for transactional replicated caching. We have introduced a language for defining and composing *invasive patterns* that has been implemented by a translation into AWED, a system for explicitly distributed AOP. Finally, we have evaluated our approach qualitatively and quantitatively by presenting a non-trivial pattern-based refactoring of parts of JBoss Cache.

Our proposal provides a solid basis for numerous future work. First, *invasive patterns* currently support static only topologies, but AWED supports groups of hosts that evolve dynamically. Our language could easily be extended to benefit from this mechanism. Second, our semantics is a simple translation to AWED so it offers many optimization opportunities (e.g., aspects deployment on specific hosts, pattern composition specialization). Finally, patterns raise abstraction level of software and are prime candidates for formal methods (properties to be analyzed include communication protocol compliance, absence of deadlock, topology invariants, fault tolerance).

References

1. Mehmet Akşit, Siobhán Clarke, Tzilla Elrad, and Robert E. Filman, editors. *Aspect-Oriented Software Development*. Addison-Wesley Professional, September 2004.
2. Deepak Alur, Dan Malks, John Crupi, Grady Booch, and Martin Fowler. *Core J2EE Patterns (Core Design Series): Best Practices and Design Strategies*. Sun Microsystems, Inc., Mountain View, CA, USA, 2003.
3. *Proceedings of the 5th ACM Int. Conf. on Aspect-Oriented Software Development (AOSD'06)*. ACM Press, March 2006.
4. AspectJ home page. `http://www.eclipse.org/aspectj`.
5. Awed home page. `http://www.emn.fr/x-info/awed`.
6. Luis Daniel Benavides Navarro, Mario Südholt, et al. Explicitly distributed AOP using AWED. In AOSD06 [3].

7. F. Buschmann, R. Meunier, H. Rohnert, P.Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley and Sons Ltd., Chichester, UK, 1996.
8. Open Management Group (OMG). CORBA components, version 3.
9. Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.
10. IBM Corp. IBM Patterns for e-business Resources. `http://www-128.ibm.com/developerworks/patterns/library`.
11. Carlos A. Cunha, João L. Sobral, and Miguel P. Monteiro. Reusable aspect-oriented implementations of concurrency patterns and mechanisms. In AOSD06 [3].
12. Rémi Douence, Didier Le Botlan, Jacques Noyé, and Mario Südholt. Concurrent aspects. In *Proc. of GPCE'06*. ACM Press, October 2006.
13. Rémi Douence, Pascal Fradet, and Mario Südholt. A framework for the detection and resolution of aspect interactions. In *Proc. of GPCE'02*, volume 2487 of *LLNCS*, pages 173–188. Springer-Verlag, October 2002.
14. J. Easton et al. *Patterns: Emerging Patterns for Enterprise Grids*. IBM Redbooks. June 2006. `http://publib-b.boulder.ibm.com/abstracts/sg246 682.html`.
15. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.
16. Jan Hannemann and Gregor Kiczales. Design pattern implementation in java and aspectj. In *Proceedings of OOPSLA'02*, pages 161–173. ACM Press, 2002.
17. JBoss Cache home page. `http://labs.jboss.com/jbosscache`.
18. Gregor Kiczales. Aspect oriented programming. In *Proc. of the Int. Workshop on Composability Issues in Object-Orientation (CIOO'96) at ECOOP*, July 1996. Selected paper published by dpunkt press, Heidelberg, Germany.
19. Gregor Kiczales, Erik Hilsdale, et al. An overview of AspectJ. In *Proc. of ECOOP'01*, LNCS 2072. Springer, June 2001.
20. Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley and Sons Ltd., Chichester, UK, 2000.
21. Stephen Siu, Mauricio De Simone, Dhrubajyoti Goswami, and Ajit Singh. Design patterns for parallel programming. In *Proc. of PDPTA'96*, volume I, pages 230–240. C.S.R.E.A. Press, August 1996. University of Waterloo, Canada.
22. Sergio Soares, Eduardo Laureano, and Paulo Borba. Implementing distribution and persistence aspects with AspectJ . In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of OOPSLA'02*, volume 37, 11 of *ACM SIGPLAN Notices*, pages 174–190, New York, November 4–8 2002. ACM Press.
23. Mario Südholt. Towards expressive, well-founded and correct Aspect-Oriented Programming. Habilitation thesis, University of Nantes, July 2007. `http://www.emn.fr/sudholt/hdr/thesis.pdf`.
24. Kai Tan, Duane Szafron, et al. Using generative design patterns to generate parallel code for a distributed memory environment. In *Proc. of PPOPP'2003*, June 2003.
25. Éric Tanter and Rodolfo Toledo. A versatile kernel for distributed AOP. In *Proceedings of the IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 2006)*, LNCS 4025. Springer, 2006.
26. Gary T. Wong, Matti A. Hiltunen, and Richard D. Schlichting. A configurable and extensible transport protocol. In *Proceedings of the 20th Annual Conference of IEEE Communications and Computer Societies (INFOCOM 2001)*, pages 319–328. IEEE, 2001.