Structured and flexible gray-box composition: Application to task rescheduling for grid benchmarking

Ismael Mejia, Mario Südholt

STRUCTURED AND FLEXIBLE GRAY-BOX COMPOSITION: APPLICATION TO TASK RESCHEDULING FOR GRID BENCHMARKING

Ismael Mejía and Mario Südholt
ASCOLA team (EMN-INRIA, LINA), Département Informatique, École des Mines de Nantes, France
{ismael.mejia, mario.sudholt}@mines-nantes.fr

ABSTRACT

The evolution of complex distributed software systems often requires intricate composition operations in order to adapt or add functionalities, react to unanticipated changes to security policies, or do performance improvements, which cannot be modularized in terms of existing services or components. They often need controlled access to selected parts of the implementation, *e.g.*, to manage exceptional situations and crosscutting within services and their compositions. However, existing composition techniques typically support only interface-level (black-box) composition or arbitrary access to the implementation (gray-box or white-box composition).
In this paper, we present a more structured approach to the composition of complex software systems that require invasive accesses. Concretely, we provide two contributions, we *(i)* present a small *kernel composition language for structured gray-box composition* with explicit control mechanisms and a corresponding aspect-based implementation; *(ii)* present and compare evolutions using this approach to gray-box composition in the context of two real-world software systems: benchmarking of grid algorithms with NASGrid and transactional replication with JBoss Cache.

KEYWORDS

Software Composition, Software Engineering, Distributed Software

1. INTRODUCTION

The evolution of large-scale distributed software systems often requires the unanticipated introduction of new functionalities or the modification of existing ones. Such evolution tasks are often inherently difficult because of two fundamental problems. First, the compositions cannot be expressed only in terms of the interfaces of the involved systems (non-invasive modifications) but also imply changes to some (typically limited) parts of the corresponding implementations. Second, the compositions often involve functionalities that are not well modularized in the existing systems or in the resulting composed system. Such composition problems occur frequently in legacy ERP systems that, *e.g.*, to cope with new security requirements imposed by changing legal frameworks, such as the Sarbanes-Oxley act in the U.S. (such evolution problems for SAP AG's SOA infrastructure are considered, *e.g.* in the CESSA[1] research project).

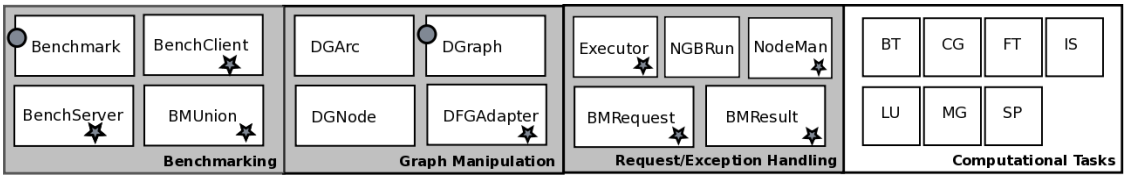


Fig. 1. NASGrid: application structure and scheduling-relevant code parts

As a concrete real-world example (that we will consider in more detail later on), we have studied NASA's NASGrid benchmarking infrastructure for computational grids (Frumkin R. et al, 2001). This benchmark is

[1] This work has been supported by the CESSA project (ANR 09-SEGI-002-01, see cessa.inria.gforge.fr).

used to time grid computations that may execute on different communication topologies. Fig. 1 shows the main components, shown with gray background, of the NASGrid benchmarking frameworks: three sets of classes that respectively provide a benchmarking interface, exception handling (principally of network conditions), and management of the graph structure representing the graph communication topology (the remainder of the system being constituted essentially by routines for numerical algorithms, called computational tasks in the figure).

NASGrid basically executes computations on distributed nodes and forwards intermediate results according to communication dependencies defined in terms of a topology graph. Grid computations are aborted in the case of exceptions, such as severe network errors; task rescheduling in the case of exceptions is not supported. We have investigated an evolution of NASGrid to add this useful functionality that fits well with existing, frequently long-running, grid applications. Our analysis of the existing code base has shown that the extension of NASGrid by task rescheduling partially requires modifications to the existing interfaces (*i.e.*, sets of public classes and methods that are marked by disks in Fig. 1). However, the extension also requires some access to the NASGrid implementation because the necessary modifications as a whole are crosscutting with respect to the existing structure of NASGrid (the corresponding classes are marked by stars in Fig. 1).

Performing such evolutions using mainstream languages or development methods is highly difficult and error prone: (i) the crosscutting nature of such evolutions involve a potentially large number of modifications that have to be carefully synchronized; (ii) structural means and semantic properties should be supported in order to control the effects of invasive modifications to implementations.

In this paper we present an approach of structured invasive, *i.e.*, gray-box, composition that supports accesses to interfaces and implementations through compositions of basic programming patterns for invasive access, resulting in gray-box compositions whose degree of invasiveness and their impact on an implementation can be controlled explicitly and flexibly. Furthermore, these operators allow crosscutting functionalities that are part of the subsystems to be expressed modularly.

Concretely, we present two contributions: First, we introduce in section 2 a kernel language for invasive composition that enables explicit and expressive compositions of invasive distributed patterns (Benavides L. et al, 2007) (henceforth simply called invasive patterns). Invasive patterns and compositions thereof provide flexible control of gray-box compositions and support the modularization of crosscutting functionalities using aspect-oriented programming techniques (Kiczales G, 1996). We also briefly present an implementation of this kernel language using the AWED system (Benavides L. et al, 2006), (AWED website, 2010) for explicitly-distributed AOP. Second, in section 3 we present and evaluate how our approach supports an evolution scenario of NASGrid that add task rescheduling. This extension is non-trivial, interacting with the original application at 28 places and is modularly implemented by an aspect and four new ordinary classes. Finally, we briefly compare the corresponding composition properties with those of two other case studies we have performed as part of previous work: a less invasive evolution of NASGrid for checkpoint introduction; and a highly invasive evolution of JBoss Cache, a middleware for transactional replication of data in distributed systems.

Our results show that invasive compositions allow a whole space of evolutions that require invasive modifications to be expressed while maintaining much higher control of the impact of invasive modifications, and this for systems requiring from moderately invasive to highly invasive accesses. As to our knowledge, no other approach to gray-box composition has been applied to such a range of evolution scenarios nor provides a comparable level of control of effects.

2. STRUCTURED AND FLEXIBLE INVASIVE COMPOSITION

Evolution scenarios as discussed in the previous section require three essential requirements to be addressed:

R1) Enable (modifications to) the coordination of distributed communications and computations.

R2) Support modularization of crosscutting functionalities that are subject to evolution tasks.

R3) Provide structural and property-based control over modifications, in particular invasive ones.

From a general point of view, we address these three issues as follows: we exploit invasive patterns as basic abstractions in order to express coordination and communication requirements of distributed applications that involve crosscutting functionalities. We introduce a composition language over patterns that

enables the definition of structured and flexible pattern compositions whose effects may be controlled, *e.g.*, by limiting invasive accesses to contexts defined by event sequences. In the remainder of this section we briefly revisit the notion of invasive patterns for distributed programming and then define a kernel language for the flexible composition of such patterns. Finally, we show how such pattern compositions can be implemented in terms of the AWED system (Benavides L. et al, 2006), a system for distributed aspects.

2.1 Invasive Patterns

Invasive patterns (Benavides L. et al, 2007) have been introduced as generalizations of standard parallel and distributed programming patterns. Fig. 2 shows the three invasive patterns we consider: a gather, a farm and a pipelining pattern. All of these patterns match sequences of execution events (illustrated by the dotted curves) over calls to interface methods or methods called in the implementation. These sequences are matched on one or several source nodes, construct data (using a computation represented by the filled rectangle on the source nodes) that is sent to a number of one or several target nodes and integrated into the computations there (as represented by the filled rectangle on the target side). Invasive patterns allow quantifying over sets of source and target nodes, in particular, the event sequences that trigger actions as well as the actions themselves.

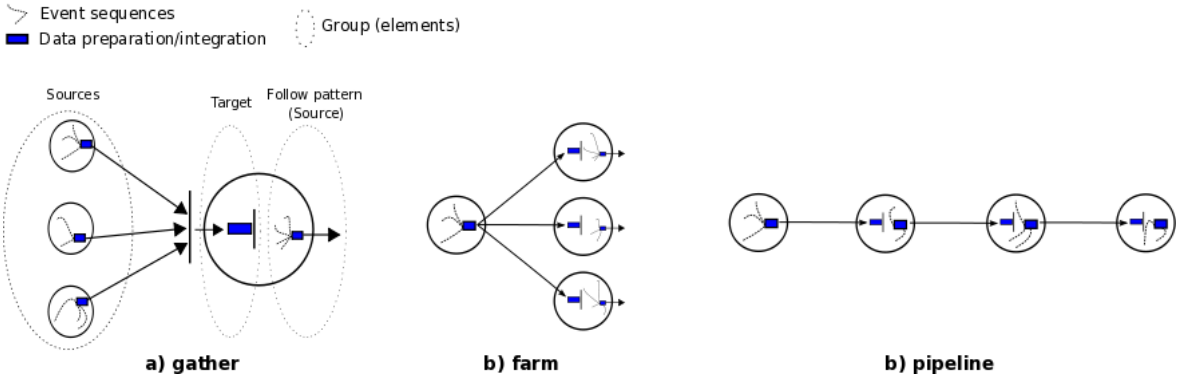


Fig. 2. Invasive Patterns

Invasive patterns provide basic support for the three requirements mentioned above: as frequently used patterns for distributed programming, they support distribution coordination (R1); modularization of crosscutting functionalities (R2) can be achieved by means of aspects for the definition of event sequences (history-based pointcuts in AOP-speak) and actions (advice in AOP-speak) that compute data to be transferred from source to target nodes and that integrate data into target computations. Finally, some control over accesses and computations is provided by their fixed overall structure.

2.2 A kernel language for non/invasive composition

In this paper, we introduce a composition language over invasive patterns in order to fully address the requirements for evolution tasks. We strive, in particular, for a language that enables flexible compositions of patterns to handle more complex crosscutting functionalities and provides better control over, possibly invasive, modifications performed by pattern compositions.

$$
\begin{array}{lcll}
Prog & ::= & \overline{Op} & \text{; Programs} \\
Op & ::= & (Ctx, Adap) & \text{; Operators} \\
Ctx & ::= & e \mid e_G \mid \overline{Ctx} & \text{; Contexts}
\end{array}
\qquad
\begin{array}{lcll}
Adap & ::= & e \mid e_G \mid P \mid \overline{Adap} & \text{; Adaptations} \\
P & ::= & (\overline{Op}, \overline{Op}) & \text{; Patterns} \\
G & ::= & \texttt{if}(B) \mid \overline{h} & \text{; Guards}
\end{array}
$$

Fig. 3. Kernel language for invasive composition

Fig. 3 presents the essentials of our kernel language for invasive composition. Some remarks on notation: non-terminals, such as *Op* or *P* start with an upper case letter and are set in italic font; lexical categories, such as *e* are denoted by lower case, italic letters; terminals, such as `if` set in typewriter font. X denotes finite

sequences of expressions of non-terminal *X*. (A more elaborate version of the language that supports, *e.g.*, repetitions in form of regular expressions is in preparation but not needed for the extension of the NASGrid application by dynamic task rescheduling considered in this paper).

The intuition behind this core language is as follows: operators match contexts that trigger sequences of adaptations. Contexts are built from event sequences that may be guarded. Adaptations include simple manipulations enabling the insertion of glue code, such as communication statements, but also potentially complex pattern compositions built from the three invasive patterns introduced above.

The grammar defines four main syntactic categories: (evolution) programs *Prog*, operators *Op*, contexts *Ctx* and adaptations *Adap*. *(Evolution) programs* are sequences over evolution operations (instantiations of invasive patterns or pattern compositions). An *operator* is defined as a pair of a context and an adaptation. *Contexts* consist of sequences of guarded events (cf. *G*), *i.e.*, events that may be matched on specific hosts or under specific conditions (represented by *B*, the nature of which is unspecified here; typically we expect conditions of limited expressiveness to support property analysis and verification).

Adaptations come in two forms: sequences of (i) possibly guarded events that represent (computation or communication) glue code potentially triggered on specific hosts and under specific conditions; (ii) structured adaptations in form of *pattern compositions P* that are pairs of sequences of operators. Such a pattern, say *(s,t)* denotes adaptations on sources *s* and targets *t*, typically the extraction of data on sources that are send for further handling to the targets. Note that patterns and pattern compositions may form both the context and adaptation parts of the operators.

Our language directly supports very flexible pattern compositions. As a simple example (that has been applied for NASGrid task rescheduling) consider an application of a farm-pattern followed by a gather pattern. The farm will match an event sequence on one node, extract information, send and inject it into a number of target nodes. The gather pattern will then monitor for event sequences on its source nodes that, in its simplest case, are the target nodes of the farm pattern, extract information on the source nodes of the gather pattern and inject them in its target node.

2.3 Implementation using distributed aspects and AWED

The AWED system (Benavides L. et al, 2006) provides an aspect model for distributed systems that provides means for the monitoring of sequences of events, history-based pointcuts in AOP-speak, that occur on different (groups of) hosts. Such event sequences are described in terms of guarded finite-state systems; AWED also provides various means to trigger actions, advice in AOP, where the corresponding pointcut-defining event sequences are matched.

The language above can be implemented using AWED in terms of event sequences that define the, interface level or implementation-level, context (*Ctx* in the above grammar) and use actions to define adaptations (*Adap*). Invasive patterns (farm, gather and pipeline) are then implemented as pairs of aspects corresponding to source and target computations of the patterns. Pattern compositions, say $p_2 \circ p_1$, are implemented by aspects that match end-marking events in p_1 and trigger execution of p_2. We have implemented task rescheduling for NASGrid using AWED this way.

Fig. 4 shows the main component of the implementation of the task rescheduling aspect. Here, the pointcut `taskRescheduling` (lines 4–11) defines a mixed interface/implementation-level context that identifies exception occurrences (state `EXCEPTION`) and, possibly repeated, choices of alternative available hosts (state `LOOKUP`). The second advice (lines 18–27) chooses an alternative and restarts the benchmark (*i.e.*, triggers a farm pattern that sends info to the successor nodes of the current one). Overall, this language provides flexible structured invasive access through pattern compositions that may be subjected to explicit control through predefined compositions and the precise definition of application contexts by means of event sequences.

Implementing composition of invasive patterns. Fig. 5 shows an interface we have developed that represents a subset of the above language that makes explicit invasive patterns and pattern compositions. The pattern composition constructors enable building of compositions from simple operators (constructor `op`), sequences of compositions (`seq`), and compositions of farm and gather patterns (the latter two being expressible as sequences and are necessary for the task rescheduling example).

```
aspect TaskReschedulingAspect perobject {
  BMRequest request;

  pointcut taskRescheduling():
    seq(
      CONFIG: call(* BenchServer.configScheduling(..)) && host(localhost) > START;
      START: call(* BenchUnion.startBenchmark(..)) && host(localhost) > EXCEPTION;
      EXCEPTION: call(* BenchServer.PutArcData(..)) && host(localhost) > LOOKUP;
      LOOKUP: call(* NodeManager.isAvailable(..)) && !host(localhost) > RESTART || LOOKUP;
      RESTART: call(* BenchServer.PutArcData(..)) && host(localhost) > START;
    );

  after() throwing: step(taskRescheduling(), EXCEPTION) {
    BenchServer serv = (BenchServer) thisJoinPoint.getCalledObject();
    request = thisJoinPoint.getArgs()[0];
  }

  after(): step(taskRescheduling(), LOOKUP) {
    NodeManager nm = (NodeManager) thisJoinPoint.getCalledObject();
    String newHost = nm.getHostId(); double loadAvg = nm.getLoadAverage(); // farm pattern
    if (evaluateHostQoS(newHost, loadAvg)) {
      DFGAdapter adapter = DFGAdapter.fromGraph(req.dfg); adapter.updateGraphDefinition(newHost);
      BenchUnion comp = new BenchUnion(req); \\ gather pattern
      comp.startBenchmark();
    } else {
     nm.lookNewNode();
    }}}
```

Fig. 4. Implementation of task rescheduling in NASGrid using AWED

```
interface InvasiveOp<Source, Target> {
    InvasiveOp<Source, Target> farm(Source src, Collection<Target> dests);
    InvasiveOp<Source, Target> pipeline(Collection<Source, Target> steps);
    InvasiveOp<Source, Target> gather(Collection<Source> origs, Target dest);
}

public interface InvasiveComp<Source, Target> {
    InvasiveComp<Source, Target> op(InvasiveOp<Source, Target>);
    InvasiveComp<Source, Target> seq(InvasiveComp<Source, Target> ops);
    InvasiveComp<Source, Target> farmGather(InvasiveOp<Source, Target> farm, InvasiveOp<Source, Target> gather);
    InvasiveComp<Source, Target> gatherFarm(InvasiveOp<Source, Target> gather, InvasiveOp<Source, Target> farm);
}
```

Fig. 5. Invasive Composition Interface

3. STRUCTURED AND FLEXIBLE INVASIVE COMPOSITION

In this section we consider the implementation of evolution scenarios using invasive patterns in the context of two real-world software systems of medium size, the NASGrid application (ca. 21 KLOC) and the JBoss Cache middleware for distributed caching under transactional control (ca. 50 KLOC). Concretely, we present the task rescheduling evolution for NASGrid in more detail, especially its use of invasive composition and how our language can be used to exert control over invasive modifications. Furthermore, we briefly compare the composition characteristics of the task rescheduling case study with two other evolution scenarios that we have previously performed using invasive patterns but without explicit support for the composition of patterns. This comparison shows that compositions of invasive patterns allow to cover a whole range of evolution scenarios, from limited invasive ones to highly invasive ones and that the flexible pattern compositions our language supports simplify such evolution tasks.

3.1 Invasive composition for NASGrid task rescheduling

NASA's NASGrid benchmark allows to time grid applications that are deployed on different hardware topologies; communication paths taken as part of a grid application are represented in NASGrid using a topology graph. Each computation on a node is modeled using an individual worker thread that executes some numerical computation, using building blocks, such as LU matrix decomposition and Fourier transforms (FT). These computational tasks are supervised by a coordinator thread which forwards the results to other nodes as defined by the topology graph.

The main obstacle for adding task rescheduling as a strategy for improving fault tolerance in the case of network or node failures, is the static topology representation and benchmark execution in NASGrid. More concretely, in terms of the NASGrid system architecture shown in Fig. 1, the graph manipulation part does not accommodate topology changes, and the benchmarking part does not allow to probe the status of network

connections, or to test the availability of remote hosts, or to modify the routing of data between nodes. Our extension to NASGrid introduces these features and exploits them when exceptional situations occur. In order to achieve that goal we have to extend the interfaces (of interfaces and classes marked disks in the diagrams) and the implementation of the classes (that are marked by stars in the figure) at multiple points. Note that a almost all disks or stars represent several modifications within the same interface or class. Overall, NASGrid has to be modified at 28 locations to extend it modularly by the task rescheduling functionality.

In order to give a concrete idea of which code manipulations are involved in invasive accesses and pattern compositions, let us first have a look at the code excerpt shown in Fig. 6. This excerpt shows the NASGrid code for localization and propagation of data between nodes. In case that a remote node is unavailable (lines 14-17) no reaction is taken and the exception is only passed along. However, as this method makes explicit the data on the real successor nodes of the current node (lines 4-8), we have to access it to update the new node information with the corresponding data after rescheduling.

```
1  public int PutArcData(BMRequest req,BMResults res)
2     throws RemoteException {
3     DGNode nd=req.dfg.node[req.pid];
4     BMRequest lreq[]=new BMRequest[nd.outDegree];
5     // ... process info
6     for(int i=0;i<nd.outDegree;i++){
7        lreq[i]=new BMRequest(req);
8        // ... clone arq info
9        try {
10          Benchmark RemBench = (Benchmark) Naming.lookup("//"+
11                lreq[i].MachineName+"/BenchmarkServer");
12          lreq[i].tmSent=System.currentTimeMillis();
13          RemBench.SendData(lreq[i],res);
14       } catch (Exception e) {
15         // ... print exception stacktrace
16         throw new RemoteException("BenchServer exception: ", e);
17 }}}
```

Fig. 6. Class BenchServer (fragment) task rescheduling

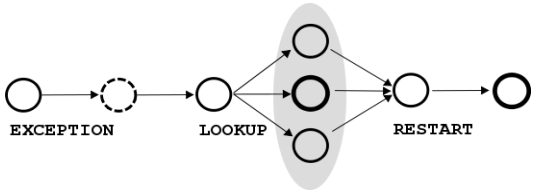


Fig. 7. NASGrid invasive composition

We have extended NASGrid with the help of invasive composition operators implemented with sequences in AWED as presented in Fig. 4. This required the extension of two interfaces: `Benchmark` to enable dynamic task rescheduling, and `DGraph` to permit the dynamic modification of the graph. We also used controlled invasive accesses to inject new code for coordination that corresponds to the identification of the precise context in which exceptional situations have to be handled by means of a sequence of events, and then a `farmGather` composition operator (cf. 5). Fig. 7 illustrates the resulting compositional algorithm (which gives a high level view of the patterns involved to modularize the task scheduling functionality): when a benchmarking execution fails (represented by the dashed circle) the exception is matched, and an execution of a farm pattern that sends a request to all successor nodes is triggered. We then use a gather pattern to collect the availability and load average information of all successor nodes and proceed to select the best available node (bold node in the figure) and reschedule the computation.

Concretely, using our AWED implementation this is performed using the rescheduling aspect shown in Fig. 4, lines 4–11, we first identify exceptional situations as a context in which a correct initialization (state `CONFIG`) and the start of a concrete benchmark (`START`) is followed by a relevant exception (`EXCEPTION`) to the method `Benchserver.PutArcData`. We then get the list of successor nodes (state `LOOKUP`) that are admissible by the topology of the grid application as defined by the user through the class `Nodemanager`. At that point, the second advice is applied (lines 18–27) that applies the `farmGather` composition in order to choose the best alternative and invasively modify the graph topology by a call to the method `adapter.updateGraphDefinition`. Finally, we restart the benchmarking operation proper (`RESTART`).

The implementation of NASGrid consists of 20490 LOC. The task rescheduling concern is implemented as a whole module using the concepts of invasive operators using 391 lines of code that correspond to the `TaskReschedulingAspect` in AWED (97 LOC) and three auxiliary classes `DFGAdapter, NodeManager` and `TaskUtility` (294 LOC). These classes perform the dynamic graph manipulation and manage the task relocation to the new nodes. Overall we therefore achieve a concise, fully modularized and compositional implementation of the extension of NASGrid by task rescheduling, Furthermore, the composition shown in Fig. 7 provides very precise control on the contexts in which invasive modifications are performed and thus

enable, in principle, to model check properties over the event sequences defining such compositions (this is however future work).

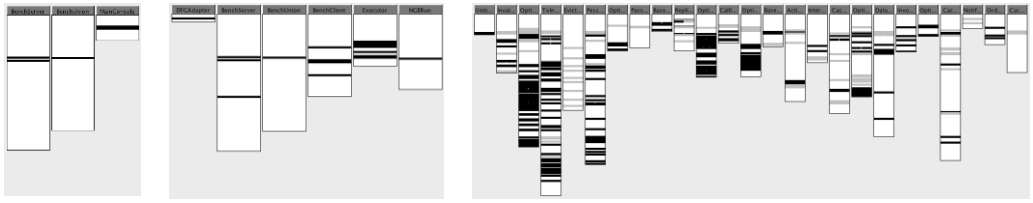
Finally, note that the overhead of task rescheduling basically consists, for each exceptional situation, in 1. A sequence of a small number of locally executed instructions up to the exceptional situation, followed by 2. A small number of parallel executions of sequences of two message exchanges for the `farmGather` composition and 3. a small number of local instructions to reschedule the benchmark). This overhead is clearly negligible compared to the execution of the benchmark itself in almost all use cases (*i.e.*, unless exceptional situations abound, a case that should very rarely constitute a reasonable application of NASGrid).

3.2 Degrees of invasive composition

In previous work we have applied invasive patterns (without support for pattern composition as introduced here) to two other evolution scenarios, an extension of NASGrid for checkpointing and an extension of the replication strategy of JBoss Cache, an infrastructure for replication under transactional control that is part of the JBoss Application Server.

Checkpointing in NASGrid. We have shown how a different reliability property of NASGrid can be improved upon via checkpointing introduction (Benavides L. et al, 2008). A checkpointing algorithm for error recovery defines a protocol to create checkpoints (snapshots of the distributed states), and guarantees the global consistency by returning to a previously-recorded state in case of failure. As we have to only add the snapshot creation and recovery actions, the degree of invasive access required is limited. It is restricted to just the context definition that triggers the snapshot creation, and also includes invasive access to the unexposed data structure in order to create its backup and play it back as part of the recovery.

Evolution of the JBoss Cache replication strategy. We have also shown how to extend the replication strategy of JBoss Cache, an infrastructure that replicates data within a cluster of distributed nodes (Benavides L. et al, 2007). The cache ensures that data replication is consistent with the transactional control over independent accesses to a distributed database. The replication and transaction functionalities are heavily crosscutting within the JBoss Cache implementation (accounting for more than 500 LOC in total scattered over a code base of around 50 KLOC). This refactoring scenario required a high level of invasive access but both concerns, replication and transactional behavior has been fully modularized using invasive patterns (once again without explicit pattern compositions).



a) NASGrid checkpointing

b) NASGrid Task rescheduling

c) JBoss Cache transactional replication

Fig. 8. Crosscutting diagrams for the three evolution case studies

Fig. 8 shows the degrees of crosscutting diagrams for the three examples. Since we have been able to perform all three evolutions in a fully modular way using invasive patterns, this provides solid evidence that our approach scales from applications that are using limited invasiveness and crosscutting to applications with concerns that are highly invasive and crosscutting.

To conclude the discussion of applications of our approach, we briefly discuss if and how we can exploit the additional control we provide through the composition language introduced in this paper. Let us consider the checkpointing introduction and replication strategy extension scenarios. In the following we briefly describe these extensions and compare how the approach presented here can improve on the previous solutions. First, the pattern applications used in the NASGrid checkpointing evolution can straightforwardly be expressed using our pattern composition language and the resulting additional control would allow, *e.g.*, to reason over the correctness properties of checkpoint-based recovery (which is, admittedly rather simple in this case anyway due to the limited invasive nature). In the case of the JBoss Cache replication evolution, the

additional control provided by our composition language is crucial in order to provide crucial correctness properties, such as the absence of certain race conditions during replication. This is also future work.

4. RELATED WORK

Our work is mainly related to three types of work: *(i)* other approaches to gray-box composition of software entities, *(ii)* approaches that use aspects in order to invasively manipulate software entities, mostly components, and *(iii)* work that advocates the use of patterns for distributed programming. None of these approaches, however, provides such flexible compositions of patterns for invasive composition that can be controlled precisely using a composition language. Because of space constraints we only discuss a few most relevant works. The probably best-known analysis of gray-box composition and an approach relying on code entities with holes as basic building blocks has been presented by (Aßmann U, 2003). Composition can be controlled by standard abstraction mechanisms such as component parameterization. This approach is however less structured than explicit pattern compositions and supports less precise control than ours. (Lorenz D et al, 2003) present a model for so-called aspectual collaboration in which aspects can be used to invasively modify software entities, mostly classes. Such aspect-based approaches provide no support in form of compositions of basic entities we do, and support only very coarse-grained control over invasive modifications. Patterns for distributed programming and massively programming (Cole M, 1989), (Schmidt D, 1996) have been mostly used as design patterns for non-invasive programming. Our previous work on invasive distributed patterns provide patterns as programming abstractions that can be composed manually but without support of a flexible composition language.

5. CONCLUSION AND PERSPECTIVES

In this paper we have made the case for more expressive and structured means for flexible gray-box compositions of distributed software systems. We have introduced a kernel language for structured flexible gray-box composition that enables to concisely define and precisely control pattern composition. We have sketched an aspect-based implementation of this language and applied it to a non-trivial extension of the NASGrid system for grid benchmarking. Finally, we have provided evidence that our approach scales from moderately to highly crosscutting applications. This work paves the way to the investigation of a (formally-defined) theory of invasive composition. In the long term, the quest for a set of operators that is complete with respect to a large number of evolution scenarios should be undertaken.

REFERENCES

Aßmann U, 2003. *Invasive Software Composition.* Springer Verlag, New York. USA

AWED website, 2010. *AWED home page* [online] available at http://awed.gforge.inria.fr (Accessed on: August 7, 2010).

Benavides L. et al, 2008. Aspect-based patterns for grid programming. *Proceedings of the 20th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'08)*. IEEE Press.

Benavides L. et al, 2007. Invasive patterns for distributed programs. *Proceedings of the 9th International Symposium on Distributed Objects,Middleware, and Applications (DOA'07)*. Vilamoura, Algarve, Portugal.

Benavides L. et al, 2006. Explicitly distributed AOP using AWED. *Proceedings of the 5th ACM Int. Conf. on Aspect-Oriented Software Development (AOSD'06)*.

Cole M, 1989. *Algorithmic skeletons: structured management of parallel computation*. Pitman.

Frumkin R. et al, 2001. Nasgrid benchmarks: a tool for grid space exploration. *High Performance Distributed Computing*, pp. 315–322.

Kiczales G, 1996. Aspect oriented programming. *Proceedings. of the International. Workshop on Composability Issues in Object-Orientation (CIOO'96) at ECOOP*. Heidelberg, Germany.

Lorenz D et al, 2003. Aspectual collaborations: Combining modules and aspects. *The Computer Journal*, Vol. 46 No. 5. pp 542–565.

Schmidt D, 1996. OO design patterns for concurrent, parallel, and distributed systems. *Proceedings of the Second USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*. Toronto, Canada