STRUCTURED AND FLEXIBLE GRAY-BOX COMPOSITION USING INVASIVE DISTRIBUTED PATTERNS [1,2]

Ismael Mejía. *ASCOLA team (EMN-INRIA, LINA), École des Mines de Nantes, France*
ismael.mejia@mines-nantes.fr

Mario Südholt. *ASCOLA team (EMN-INRIA, LINA), École des Mines de Nantes, France*
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, to react to unanticipated changes, or to apply performance improvements that cannot be modularized in terms of existing services and components. These evolutions 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 structured approach to the composition of complex software systems that require invasive modifications. Concretely, we provide three contributions: (i) we present a *small kernel composition language* for structured gray-box composition using invasive distributed patterns; (ii) we motivate that gray-box composition approaches should be defined and evaluated in terms of the *flexibility and control* they provide, a notion of degrees of invasiveness is introduced to help assess this trade-off; (iii) we apply our approach to a new case study of evolution and evaluate it in the context of two previous studies involving *two real-world software systems*: benchmarking of grid algorithms with NASGrid and transactional replication with JBoss Cache.
As a main result, we show that *gray-box composition using invasive distributed patterns* allows the declarative and modular definition of evolutions of real-world applications that need moderate to high degrees of invasive modifications.

KEYWORDS

Software Composition, Software Engineering, Distributed Software, Aspect Oriented Programming, Grid Computing, Middleware

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

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

[2] A preliminary version of this article was presented at the IADIS International Conference on Applied Computing 2010 (Mejia, 2010).

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 research project).

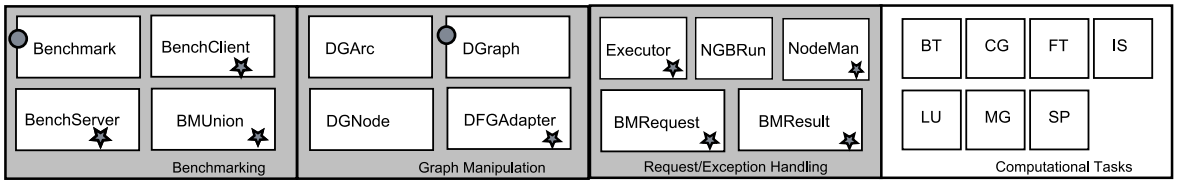


Fig. 1. NASGrid: application structure and scheduling-relevant code parts

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 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 three contributions: First, we introduce a kernel language for invasive composition that enables explicit and expressive compositions of invasive distributed patterns (Benavides L. et al, 2008) (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, we present and evaluate how our approach supports an evolution scenario that adds task rescheduling to NASGrid. This extension is non-trivial and interacts with the original application at 28 places. We show how to modularly implement it by an aspect and four new ordinary classes.

Third, we motivate that gray-box composition should provide flexibility but also strong (structural and property-based) control over invasive modifications. Control is particularly important in order to provide correctness guarantees in the presence of crosscutting concerns. We introduce the notion of degrees of invasiveness. These degrees enable complex compositions to be assessed with respect to their needs for invasive modifications.

Our results show that invasive patterns allow a whole space of evolutions to be expressed that require moderate to high degrees of invasive modifications. Furthermore, compared to previous work they support stronger control over the impact of invasive modifications. This result is backed by an analysis of three evolutions of real-world applications: apart from NASGrid task rescheduling, we discuss two other case studies that we have performed as part of previous work: a less invasive evolution of NASGrid for checkpoint introduction; and a much more invasive evolution of JBoss Cache, a middleware for transactional replication of data in distributed systems.

As to our knowledge, no other approach to gray-box composition has been applied to such a range of evolution scenarios and provides enough flexibility and control to fully modularize the necessary changes.

The article is structured as follows: Section 1 motivates gray-box composition problems in the context of the NASGrid benchmarking application. We define our kernel language for invasive composition in section 2. In section 3 we apply our method to introduce task rescheduling into NASGrid. The characterization of gray-box composition in terms of flexibility and control is presented in section 4. Section 5 presents related work and section 6 concludes.

2. STRUCTURED AND FLEXIBLE INVASIVE COMPOSITION

While gray-box composition is a well-known and widely-used concept, it is almost always realized in the form of principles and constraints on the use of general-purpose mechanisms which may modify architectures and applications in an unrestricted manner. Frequently, gray-box compositions are therefore implemented using sequences of low-level refactoring transformations or even in terms of arbitrary modifications using general-purpose programming languages. Furthermore, they are rarely supported by higher-level design methods or guided by pattern-based composition strategies. This approach allows arbitrary compositions to be applied but provides only very few structure to help understanding of the modifications. Furthermore, the correctness of the resulting modifications can only be ascertained with much difficulty. Finally, most existing approaches provide little support for crosscutting functionalities, such as task rescheduling in the NASGrid application that cannot be defined concisely and modularly using traditional notions of, *e.g.*, objects and components.

In this section, we provide structured language-level support extended the approach of invasive patterns for invasive composition. We first introduce and discuss several high-level requirements that approaches for invasive composition should meet. We then give necessary background information on invasive patterns before introducing our new kernel language and a first implementation. Finally, we revisit our approach in light of the initial requirements.

2.1 Requirements

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

R1) *Enable flexible modifications to the coordination of distributed communications and computations* Note that we are less interested in arbitrary refactoring but rather in the coordination of activities in a comprehensive sense, notably message exchanges, remote events and remote execution of activities, as well as the scheduling of activities.

R2) *Support modularization of crosscutting functionalities that are subject to evolution tasks* Crosscutting functionalities are of paramount importance in most large-scale software systems, notably distributed ones. We therefore strive for an evolution method that allows to smoothly integrate them.

R3) *Provide structural and property-based control over modifications, in particular invasive ones* Structural control enables modifications to be defined in a modular way (with respect to the original software structure). Property-based control permits the definition of modifications in terms of the execution history of computations, including predicates on the current execution state.

As to the best of our knowledge, a fair number of previous approaches supports R1 or R2 (the latter being limited to AO or reflective ones) but none supports all three requirements.

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.

2.2 Invasive patterns

Invasive patterns (Benavides L. et al, 2008) have been introduced as generalizations of standard parallel and distributed programming patterns. Fig. 2 shows three invasive patterns we consider: a gather, a farm and a pipelining pattern. All of them 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 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.

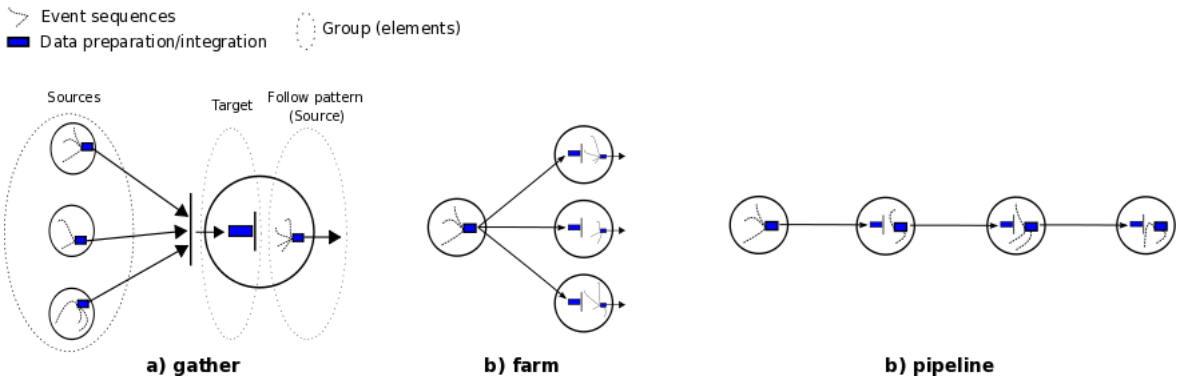


Fig. 2. Invasive Patterns

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 point-cuts 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.

2.3 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.

$$
\begin{array}{lcll}
Prog & ::= & \overline{P} & \text{; Programs} \\
P & ::= & (Ctx, Adap) & \text{; Patterns}
\end{array}
$$

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

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 *Prog* 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 the following syntactic categories: (evolution) programs *Prog*, patterns *P*, contexts *Ctx*, adaptations *Adap*, and guards *G*. *(Evolution) programs* are sequences over evolution operations (instantiations of invasive patterns or pattern compositions). A *pattern* is defined as a pair of a context and an adaptation. 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.

Contexts basically of sequences of guarded events, *i.e.*, events that may be matched on specific hosts (represented by subscript *h*) 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 basically 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.

Note that patterns and pattern compositions may form both the context and adaptation parts of operators. Patterns used in a context position define (possibly guarded) event sequences where an adaptation takes place. Patterns in context position may also perform adaptations themselves: this is used, *e.g.*, to prepare information that is extracted from the context and has to be sent to another host as part of the adaptation (of the encompassing pattern definition). Similarly, patterns as adaptations are typically used to define modifications to be performed locally on the target of the encompassing pattern definition.

Examples. As a first simple example in the context of task scheduling, consider the problem that we have to determine exceptional states at a given node and prepare corresponding summary information to be used for task rescheduling. The exceptional situation may only be relevant for NASGrid task rescheduling in specific contexts and not generally on the occurrence of a given event. This can be modeled by a pattern of the following form

$$(es, summarize(es)) \quad \text{where} \quad es = \langle e^1, \ldots, e^n \rangle$$

is a sequence of events that characterizes the interesting exceptional situations and summarize calculates the summary information based on the past events. Note that, in this example, all events happen on the same node but in general context definitions may refer to remote events.

Our language supports powerful pattern compositions. As a second example in the context of task rescheduling, consider that we need to farm out inquiries about potential exceptional situations to successor nodes and gather the resulting summary information in order to decide on concrete rescheduling actions. This can be defined application of a farm pattern followed by a gather pattern. The farm will match an event sequence on the orginating node, extract information that is needed to identify relevant exceptional situations, send and inject it into the successor nodes. The gather pattern will then monitor for relevant exceptional situations on all successor nodes, summarizes the information on the successor nodes, returns and inject the information on the originating nodes. This algorithm can be defined using our kernel language as follows:

$$\langle farmP,\ gatherP \rangle \quad \text{where}$$

$$farmP \ := (identify_o(\overline{e_o^1}),\ inject_{\bar{s}}(handleException_{\bar{s}})$$
$$gatherP := (summarize_{\bar{s}}(\overline{e_{\bar{s}}^2}),\ inject_o(decideRescheduling_o)$$

This definition makes explicit the composition of the two patterns *farmP; gatherP*. The pattern definitions are expressed in terms of the execution events relevant for task rescheduling

$$\overline{e_o^1},\ handleException_{\bar{s}},\ \overline{e_{\bar{s}}^2},\ decideRescheduling_o$$

all events are indexed by the hosts they occur on, either the original one o or each host from the set of successors s) and calls to three auxiliary methods $identify_o$; $inject_s$; $summarize_s$ (all applications of these methods are also located either on the original node or all successor nodes). Note that in this example, the

methods are executed on the same hosts on which the information they depend on (*i.e.*, the arguments to the methods) is found: this is not the case in general.

2.4 Implementation using AWED

The AWED system provides an aspect model for distributed systems that provides means for the monitoring of sequences of events, history-based point-cuts 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 point-cut 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, 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.

```
1  aspect TaskReschedulingAspect perobject {
2    BMRequest request;
3
4    pointcut taskRescheduling():
5      seq(
6        CONFIG: call(* BenchServer.configScheduling(..)) && host(localhost) > START;
7        START: call(* BenchUnion.startBenchmark(..)) && host(localhost) > EXCEPTION;
8        EXCEPTION: call(* BenchServer.PutArcData(..)) && host(localhost) > LOOKUP;
9        LOOKUP: call(* NodeManager.isAvailable(..)) && !host(localhost) > RESTART || LOOKUP;
10       RESTART: call(* BenchServer.PutArcData(..)) && host(localhost) > START;
11     );
12
13   after() throwing: step(taskRescheduling(), EXCEPTION) {
14     BenchServer serv = (BenchServer) thisJoinPoint.getCalledObject();
15     request = thisJoinPoint.getArgs()[0];
16   }
17
18   after(): step(taskRescheduling(), LOOKUP) {
19     NodeManager nm = (NodeManager) thisJoinPoint.getCalledObject();
20     String newHost = nm.getHostId(); double loadAvg = nm.getLoadAverage(); // farm pattern
21     if (evaluateHostQoS(newHost, loadAvg)) {
22       DFGAdapter adapter = DFGAdapter.fromGraph(req.dfg); adapter.updateGraphDefinition(newHost);
23       BenchUnion comp = new BenchUnion(req); \\ gather pattern
24       comp.startBenchmark();
25    } else {
26      nm.lookNewNode();
27    }}}
```

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

Fig. 4 shows the main component of the implementation of the task rescheduling aspect. Here, the point-cut `taskRescheduling` (lines 4–11) defines a mixed interface/implementation-level context that identifies exception occurrences (state `EXCEPTION`) and, possibly multiple, 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.

```
1  interface InvasiveOp<Source, Target> {
2      InvasiveOp<Source, Target> farm(Source src, Collection<Target> dests);
3      InvasiveOp<Source, Target> pipeline(Collection<Source, Target> steps);
4      InvasiveOp<Source, Target> gather(Collection<Source> origs, Target dest);
5  }
6
7  public interface InvasiveComp<Source, Target> {
8      InvasiveComp<Source, Target> op(InvasiveOp<Source, Target>);
9      InvasiveComp<Source, Target> seq(InvasiveComp<Source, Target> ops);
10     InvasiveComp<Source, Target> farmGather(InvasiveOp<Source, Target> farm, InvasiveOp<Source, Target> gather);
11     InvasiveComp<Source, Target> gatherFarm(InvasiveOp<Source, Target> gather, InvasiveOp<Source, Target> farm);
12 }
```

Fig. 5. Invasive Composition Interface

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).

2.4 Requirements revisited

We are now ready to evaluate our approach in the light of the requirements introduced above.

R1) Patterns directly allow communications and computations to be coordinated as the passing of information and synchronization between the context and adaptation components of patterns.

R2) The modularization of crosscutting functionalities is also directly supported through the quantification over events within the context and adaptation definitions as well as the indexing of events and pattern definitions over sets of hosts. Furthermore, the context and adaptation components of a pattern are similar to the point-cut and advice of traditional aspect languages.

R3) Structural and property-based control over invasive compositions is supported by means of the flexible constraints that can be harnessed as part of pattern definitions, *e.g.*, in terms of sets of hosts and condition guards.

Our composition approach meets the three requirements.

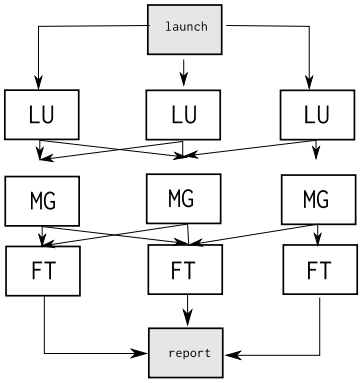
3. APPLICATION OF FLEXIBLE INVASIVE COMPOSITION FOR SOFTWARE EVOLUTION

Middleware, *i.e.*, typically component-based software that mediates between a (potentially) distributed operation systems and applications, is a popular approach to cope with the increasing complexity of the development of software systems.Middleware has been accepted by the Industry as a core component of programs that range from web applications to complex data processing architectures, which forms out a core element of multiple applications like financial solutions, enterprise information systems, complex scientific applications, etc. Middleware is frequently subject to evolution because changes in the requirements at the system and application levels may also require changes to the mediating abstraction.

In this section we consider the implementation of an evolution scenario realized using invasive patterns in the context of a real-world middleware system of medium size, the NASGrid application (ca. 21 KLOC). Concretely, we present an evolution scenario which introduces task rescheduling to NASGrid: task rescheduling allows to continue execution of a grid benchmark in case that an originally planned pathway is no longer available. We especially study the use of invasive composition for task rescheduling and how our language can be used to exert control over invasive modifications.

3.1 NASGrid: A grid benchmarking application

We first briefly introduce the application which is a target for the evolution scenario: NASA's NASGrid benchmark for benchmarking executions of grid applications. Grid applications execute complex computations in heterogeneous network topologies. NASGrid supports the formulation of grid algorithms in terms of four different patterns akin to those for massively parallel programming: Helical Chain (HC), Visualization Pipe (VP), Mixed Bag (MB, see figure 6 left) and Embarrassingly Distributed (ED). In order to model different grid algorithms, NASGrid permits to define a data flow graph (DFG, see figure 6 right) that represents the different computations to be executed in the network as well as the (asynchronous) data flow between them. Each computation on a node is modeled by an individual worker thread that executes some numerical computation using common grid algorithm building blocks, such as LU matrix decomposition and Fourier transforms (FT). These computational tasks are supervised by a coordinator thread which sends the results to other nodes as defined by the topology graph.




```
1   graph: {
2     title: "MB.S"
3     node:{title: "0" label: "LUTask.S_16.awednet1"}
4     node:{title: "1" label: "LUTask.S_12.awednet1"}
5     node:{title: "2" label: "LUTask.S_10.awednet1"}
6     node:{title: "3" label: "MGTask.S_1.awednet1"}
7     node:{title: "4" label: "MGTask.S_1.awednet1"}
8     node:{title: "5" label: "MGTask.S_1.awednet1"}
9     node:{title: "6" label: "FTTask.S_2.awednet1"}
10    node:{title: "7" label: "FTTask.S_1.awednet1"}
11    node:{title: "8" label: "FTTask.S_1.awednet1"}
12    edge:{sourcename: "0" targetname:"3"}
13    edge:{sourcename: "0" targetname:"4"}
14    edge:{sourcename: "1" targetname:"3"}
15    edge:{sourcename: "1" targetname:"4"}
16    edge:{sourcename: "2" targetname:"4"}
17    edge:{sourcename: "2" targetname:"5"}
18    edge:{sourcename: "3" targetname:"6"}
19    edge:{sourcename: "3" targetname:"7"}
20    edge:{sourcename: "4" targetname:"6"}
21    edge:{sourcename: "4" targetname:"7"}
22    edge:{sourcename: "5" targetname:"7"}
23    edge:{sourcename: "5" targetname:"8"}
24  }
```

Fig. 6. Pattern (left) and topology definition (right) used in NASGrid

Fig. 6 (left) represents the dataflow of a mixed bag problem, and its textual representation as a data flow graph in NASGrid (on the right). A program like this is commonly executed to measure the performance of a grid architecture. However, there are many aspects that NASGrid ignores, *e.g.*, how to react if a node is down, how to save the state during a computation, or how to measure those additional tasks: all of these are common problems that grid infrastructures must handle to be flexible enough to handle, *e.g.*, common exceptional situations in a distributed system.

3.2 Invasive composition for NASGrid task rescheduling

Task rescheduling is a common strategy to achieve high availability in fault-tolerant systems. In a nutshell, it consists in the capacity to reschedule tasks, *e.g.*, in the case of errors or for purposes of load balancing, in order to correct or optimize distributed applications.

The main obstacle for adding task rescheduling in NASGrid, is the static topology representation and benchmark execution. 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, nor to test the availability of remote hosts, nor to modify the routing of data between nodes. Our extension to NASGrid introduces these features and exploits them when an exceptional situation 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. 7. 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. 7. Class BenchServer (fragment) task rescheduling

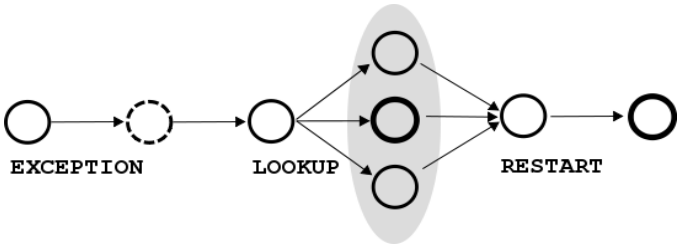


Fig. 8. NASGrid invasive composition

We have extended NASGrid by using 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. 8 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, then we 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 of the run 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. 8 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 happen in a reasonable application of NASGrid).

4. GRAY-BOX COMPOSITION: FLEXIBILITY VS. CONTROL

Black-box composition is a desirable property for a composition framework because it enables a strict separation of duties in the implementation, use and maintenance of components. These properties only apply to a lesser degree when gray-box composition mechanisms are used (and even not at all in the borderline case where gray-box composition meets white-box composition). Since invasive composition is frequently needed for evolution tasks, in particular, of legacy software systems, it is therefore important to be able to estimate (i) when invasive composition is technically and practically feasible, and (ii) to what extent the desirable properties of black-box composition hold. However, no corresponding studies exist and the existing literature on gray-box composition (notably (Assmann U., 2003), (Kellner K. et al, 2000), (Kojarski S. et al, 2005)) addresses these questions rudimentarily at best.

A systematic treatment of the trade-off between flexibility of the compositions and properties such as implementation independence and provision of correctness guarantees has to be based on main parameters:

- Qualitative and quantitative information of the degree of invasiveness that is needed, in particular, in the context real-world applications.
- An analysis of existing composition approaches with respect to the flexibility of and the control over the invasive modifications they provide, as well as the definition of corresponding new composition mechanisms.

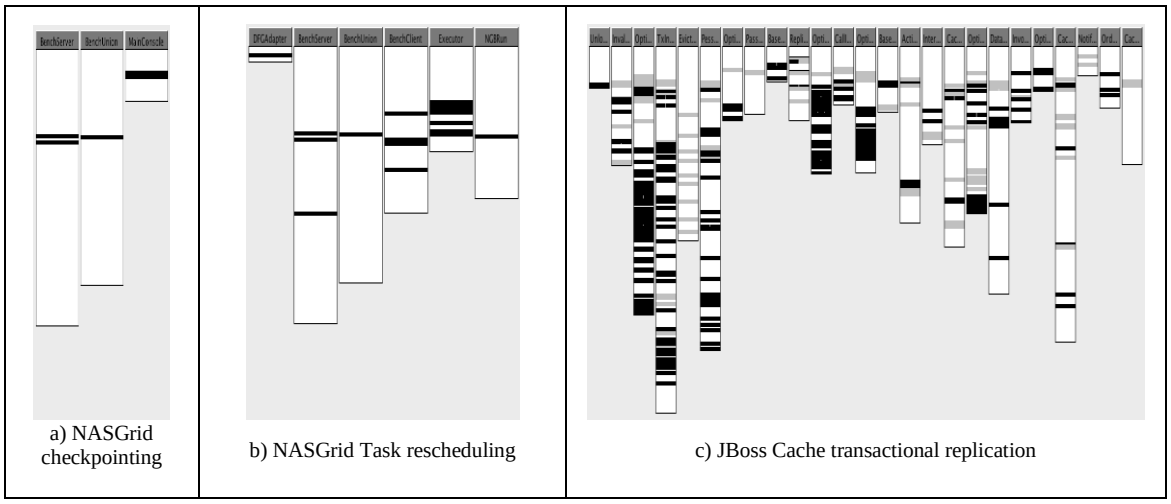
In the following we present the first analysis (to the best of our knowledge) of the application of gray-box composition techniques that require a widely different degrees of invasive access.

4.1 Analysis of three real-world evolution scenarios

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 injection 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 the introduction of checkpoints (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 the case of failure. As we only have to 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, it 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), (Benavides L. et al, 2008). 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. 9. Crosscutting diagrams for the three evolution case studies

Degrees of invasiveness. Fig. 9 shows crosscutting diagrams for these two scenarios and the Task rescheduling case study presented in this paper. In the three diagrams columns represent Java classes and stripes within the columns represent the code that is relevant for the corresponding crosscutting functionality: code relevant for checkpointing injection in 9a, code necessary for the definition of task rescheduling in part b, and code implementing transactions or replication functionality in part c.

These figures clearly show that the three case studies require compositions of largely different degrees of invasive access. The checkpointing evolution requires very limited invasive access, the task rescheduling one a moderate number of accesses, and the replication evolution a large number of invasive accesses. Furthermore, none of the three evolutions can be implemented using exclusively black-box composition, and the latter two scenarios are difficult to perform with traditional (white-box) programming mechanisms.

4.2 Analysis of the use of invasive patterns

We can now evaluate the use of invasive patterns in these three evolution case studies with respect to the main trade-off flexibility (of compositions) vs. control (that supports in particular correctness guarantees).

Overall this analysis provides solid evidence that invasive patterns provide, in contrast to all previous approaches to gray-box composition, sufficient flexibility and reasonable control to accommodate all degrees of invasiveness required in real-world applications.

Flexibility We have been able to perform all three evolutions in a fully modular way using invasive patterns, thus providing solid evidence that our approach is flexible enough to support applications which require limited invasiveness but also those applications with highly invasive and crosscutting concerns.

Control Our approach provides control over gray-box compositions through the declarative definitions of complex evolutions using pattern compositions. In the case of task rescheduling, the pattern compositions define, for instance, how communications and computations have to be coordinated. Furthermore, the context definitions in patterns define precisely which (potentially crosscutting) execution events are relevant for gray-box compositions.

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 the principal correctness properties, such as the absence of certain race conditions during replication. This is however subject of future work.

5. RELATED WORK

Our work is mainly related to three types of work:

- Other approaches to gray-box composition of software entities.
- Approaches that use aspects in order to invasively manipulate software entities, mostly components.
- Work that proposes the use of patterns for distributed programming.

Compared to our approach, none of these approaches provides similar flexible gray-box compositions that can be controlled precisely using a composition language.

5.1 Gray-box composition of software entities

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. However, this approach is less structured than explicit pattern compositions and supports less precise control than ours. Work on aspect-aware interfaces (Kiczales G. et al, 2005) and open modules (Aldrich J., 2005) provides means to export restricted invasive capabilities in component systems without affecting important properties of modular reasoning. The work by (Kellner K. et al, 2000) discusses approaches for the construction of business processes in which different applications can be composed using black, gray, glass or white box components. Our approach, as the first one is equally suited for moderate to highly invasive modifications, but we include a methodology that combines invasiveness and distribution.

5.2 Invasive modifications using aspects

The work of (Lorenz D. et al, 2003) presents 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 structured support in the form of compositions of basic entities as we do and support only very coarse-grained control over invasive modifications. Kojarski and Lorenz (Kojarski S. et al, 2006) study the invasive composition of aspect mechanisms in the context of interacting requirements. They compare the interactions between an aspect for fault-tolerance and one for access-control written in different languages and with different degrees of invasiveness. Our work similarly relies on the interaction of elements with invasive capabilities. However, our language also offers expressive mechanisms to compose different patterns in order to define advanced control flows and control invasive compositions.

5.3 Patterns for distributed programming

Cole's work on algorithmic skeletons (Cole, 1989) has introduced the representation of common design solutions for the implementation of principally massively-parallel algorithms via patterns who abstracted common operations like pipelines, farms, etc. Our work can be seen as providing a generalization of these patterns for the programming of heterogeneous distributed algorithms. (Schmidt D., 1996) has proposed a new set of common patterns for distributed programming, in particular for event-based programming. Other group of patterns have being studied in specific contexts, *e.g.*, for distributed workflow systems (Van der Aarst W. et al, 2003), and in the space of synchronous/asynchronous for the integration of enterprise systems (Hohpe G. et al, 2003). All these previous approaches for patterns for distributed systems define models for non-invasive programming. Our previous work on invasive patterns (Benavides L. et al, 2008) provide patterns as programming abstractions that can be composed manually but without the support of a flexible composition language. Our current work expands invasive patterns with a more rigorous framework that includes a concrete language to define compositions. Furthermore, we have introduced degrees of invasiveness to clarify the benefits of our approach over a range of applications with widely-different gray-box composition characteristics

6. CONCLUSION AND PERSPECTIVES

In this paper we have studied invasive distributed patterns as a means to precise, define and implement gray-box compositions for the moderately to highly invasive evolution of real-world distributed applications and middleware, in particular to satisfy a set of common requirements for the evolution of such systems. We have presented three contributions: (i) a kernel language for structured and controlled flexible gray-box composition, (ii) an application of this kernel language and a corresponding implementation using AWED, a system for distributed AOP, to the introduction of task rescheduling into NASA's grid benchmarking application; (iii) a characterization of gray-box composition in terms of flexibility and control, the evaluation of which is supported by a notion of degrees of invasiveness.

This work paves the way to the investigation of a (formally-defined) theory of gray-box composition. Such a theory should extend the structural and property-based control over flexible pattern compositions as presented in this article. Furthermore, it should be complemented by methods for the proof of properties of gray-box compositions, *e.g.*, non-interference properties as well as properties of synchronization and sequentialization of pattern compositions. Finally, the quest for a set of operators that is complete with respect to a large number of evolution scenarios should be undertaken.

REFERENCES

Aldrich J, 2005. Open modules: Modular reasoning about advice. *Proceedings of European Conference of Object Oriented Programming*. Glasgow, United Kingdom. pp. 144–168.

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, 2008. Debugging and testing middleware with aspect-based control-flow and causal patterns. *Proceedings of the ACM/IFIP/USENIX 9th International Middleware Conference*. Leuven, Belgium, pp. 183–202.

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.

Fellner K. et al, 2000. Classification framework for business components. *Proceedings of the 33rd Annual Hawaii International Conference on (HICSS'00)*.

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

Hohpe G. et al, 2003. *Enterprise Integration Patterns: Designing, building, and deploying messaging solutions.* Addison-Wesley. Boston, MA, USA.

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

Kiczales G et al, 2005. Aspect-oriented programming and modular reasoning. *Proceedings of the 27th international conference on Software engineering, ICSE '05*, pages 49–58, New York, NY, USA.

Kojarski S. et al, 2006. Comparing white-box, black-box, and glass-box composition of aspect mechanisms. *Proceedings of the. 9th International Conference on Software Reuse, ICSR 2006*, Turin, Italy, pp. 246–259.

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

Mejía I. et al, 2010. Structured and flexible gray-box composition: application to task rescheduling for grid benchmarking. *Proceedings of the IADIS International Conference on Applied Computing 2010*, Timisoara, Romania.

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

Van der Aalst, et al. 2003. Workflow patterns. *Distributed and parallel databases*, Vol. 14 No. 1, pages 5–51.