# Invasive composition for the evolution of a health information system

Ismael Mejía, Mario Südholt
ASCOLA group; EMN-INRIA, LINA
Dépt. Informatique. École des Mines de Nantes
4 rue Alfred Kastler, 44307 NANTES Cedex 3, France
First.Last@mines-nantes.fr

Luis Daniel Benavides Navarro
Grupo de Construcción de Software
Departamento de Ing. de Sistemas
Universidad de Los Andes
Cra 1 No. 18A- 12, Bogotá, Colombia
lbenavidesnavarro@acm.org

## ABSTRACT

In this paper we show that some of the evolution tasks in OpenMRS, a health information system, may require the invasive modification of interfaces and implementations in order to offer an appropriate modularization. We introduce a new composition framework in Java that supports the definition of expressive pattern-based invasive compositions. Furthermore, we show that the composition framework allows us to concisely define an evolution scenario of OpenMRS that supports the consolidation of patient data from different remote instances.

## Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Frameworks, Patterns*

## General Terms

Design, Languages

## Keywords

Aspect-oriented programming, Distributed systems, Health information systems, Invasive software composition

## 1. INTRODUCTION

Evolution of large-scale distributed systems is a fundamental challenge of current information systems, notably web-based ones. Two problems are particularly difficult to handle:

- The modification of heterogeneous distribution and communication requirements, for example, if previously colocated parts of an application are to be executed in a remote location.
- The need for invasive modifications [1], *i.e.,* modifications to implementations, *e.g.,* to enrich composition interfaces by exposing previously hidden behaviors.

We have studied these problems in the context of an information system for the health domain, the OpenMRS system [11], an open-source web-based medical record management system[1]. In this paper we consider an evolution of this system that enables the consolidation of information of a patient's medical record history from different sources. Such consolidation is needed, for instance, if different instances of OpenMRS are used in different locations. This system and evolution scenario is subject to the two problems mentioned above: the consolidation relies on data gathered from different sources (*i.e.,* requires modification of communication requirements) and requires modifications to the handling of a patient's medical history that is not expressible in terms of the OpenMRS's corresponding interfaces but need (limited) access to implementations (requires invasive modifications to enrich composition interfaces).

Traditionally, such evolutions are implemented "by hand" without tool support for the required structural and behavioral modifications. Existing tools are limited to the generation of parts of the implementation from higher-level specifications, *e.g.,* using model-driven engineering, or low-level manipulation of implementations using refactoring techniques. Frequently there is no higher-level specification to derive implementations from and evolutions are only expressible in terms of many non-trivial and low-level program refactorings. The evolution scenarios of OpenMRS that we consider are of this kind. Furthermore, design patterns and implementation patterns that provide, in principle, a general solution to such kinds of evolution scenarios cannot be applied easily to distributed applications that have heterogeneous communication and computation requirements (in contrast to more regular or 'simpler' systems, such as massively parallel applications [4], workflow definitions [14], or system integration in terms of message manipulations [7]).

Our approach to support heterogeneous evolutions scenarios builds on pattern-like solutions like algorithm skeletons [6] for parallel algorithms. The corresponding patterns include FARMING OUT the same computation on many code or a GATHER pattern that blocks the execution of a task until it has received a set of responses from different nodes. *Invasive distributed patterns* [2] extend algorithmic skeletons by the

[1]In this paper we have analyzed and applied an evolution task to OpenMRS, version 1.7.


*Varicomp '11* March 21-25, 2011, Pernambuco, Brazil

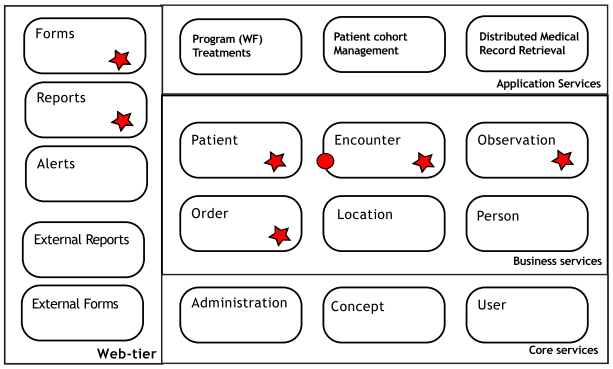


**Figure 1: OpenMRS Architecture Diagram**

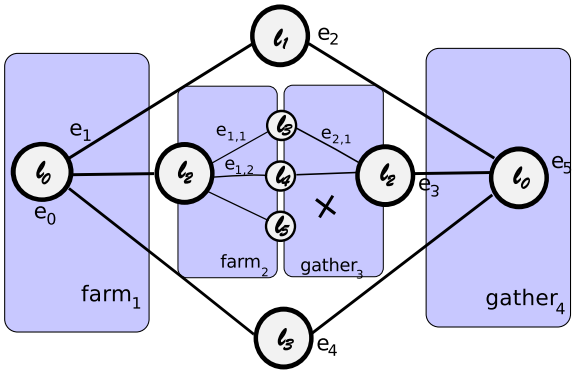


**Figure 2: Pattern compositions for electronic medical record retrieval**

ability to access component implementations to detect and modify previously unknown information about its state and behavior. They enable the description of explicitly complex communication patterns that are commonly hidden at the code level.

In this paper, we sketch an extension to the approach of invasive patterns [2, 9] that allows evolutions to be defined in terms of a language for, potentially invasive, composition. Concretely, we present the following contributions and issues for discussion:

- We briefly introduce a first design and implementation in Java of a composition framework for invasive patterns that instantiates the abstract language defined in [9].
- We motivate an evolution scenario of OpenMRS in which invasive composition is appropriate. We show how to implement it using our the concept of invasive patterns with our composition framework.

## 2. CONSOLIDATION OF ENCOUNTER INFORMATION IN OpenMRS

We have investigated evolution problems in the context of health information systems (HIS). Non-anticipated evolution tasks occur frequently in HIS when new technologies or applications need to be integrated, as well as when new business needs appear (*e.g.,* new treatments, new legislation rules or new administrative processes).

We are investigating such evolution tasks in the context of the OpenMRS HIS [11], a well known HIS created to track medical records (patients, visits, diseases, etc) that has been deployed and used in several communities, mostly in developing countries. We present results related to one evolution task in this paper: enriching medical encounter information of patients with information concerning that patient from different sources, *i.e.,* remote instances of OpenMRS where the patient has been treated. Such consolidation is useful *e.g.,* to implement strategies to identify and handle epidemic situations. Technically, this task requires the addition of functionality for the management of distribution between different OpenMRS instances and consolidating patient information in the resulting distributed system (in the remainder, we denote this task by ConsolPatData).

Fig. 1 shows a representation of the multi-tier architecture of OpenMRS. The ovals in the figure denote individual classes that represent the main functionalities of the system: patient and location management, consultations (aka medical encounters) with their respective diagnostic findings (aka observations) and orders, as well as administrative tasks like the creation of users, concepts (*e.g.,* new drugs) and treatments. The part of the implementation relevant to our study can be roughly structured in two parts (we do not consider the data tier, *i.e.,* how data is stored): the implementation of the web-tier and the health business functionality. The latter is implemented based on several core services, the business services, and an application-level service.

The implementation of the ConsolPatData evolution task basically requires two sets of modifications: (i) new functionality has to be inserted for the exchange of information between distributed instances of OpenMRS and (ii) the existing data handling processes have to be modified in order to correctly integrate the new information. Note that the latter requirement does not mean to only store the remote data and then handle it as local one: the fact that remote and local data co-exist means that existing procedures have to be modified, *e.g.,* to make decisions only after the evaluation of the patient's health records at the remote and local site have been analyzed and compared.

In Fig. 1 gray stars and circles mark the parts of the OpenMRS implementation that are affected by these two sets of modifications. The stars mark invasive modifications, *i.e.,* modifications to the implementations of the corresponding functionalities; the circle marks an interface-level modification.

The ConsolPatData evolution can be represented by the pattern composition shown in Fig. 2. Read from left to right, it defines how the information of a patient's medical record history is consolidated. First a medical center farms out (by means of pattern $farm_1$) an execution request to different nodes, *e.g.,* other hospitals and clinics, in order to get the encounter information corresponding to a patient. Then some of the institutions relay requests to some of its internal departments ($farm_2$). As some of the departments do not have information for the patient we allow for partial compositions using a partial gather pattern ($gather_3$). Finally, the answer of all the hosts is consolidated using the gather pattern $gather_4$ to the requesting facility in order to construct the distributed medical record .

Implementing this pattern composition requires invasive modifications to be performed: in the initial OpenMRS in-

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

$e$ : (Located) Events

**Figure 3: Kernel language for invasion composition**

stance patient data processes have to be modified in order to integrate and handle the remote data, in the other instances the information requested from the initial instance has to be extracted in order to be transformed. Furthermore, in all instances distribution code has to be introduced (since OpenMRS does not handle distribution natively). This composition task therefore cannot be performed solely in terms of standard patterns for distribution, such as described in [14, 7].

The necessary modifications are non-trivial and have to be applied at 16 different source code locations (and partially multiple times in different execution contexts at those locations). In previous work we have introduced invasive distributed patterns [2, 9] to handle such evolution tasks. That work introduced the conceptual framework and applied it using ad hoc implementations of pattern compositions. Here, we extend our previous work by presenting a first version of a Java framework for the systematic composition of invasive patterns. In the remainder of this paper, we briefly present this framework and illustrate its benefits for the evolution of OpenMRS.

# 3. A LANGUAGE FOR INVASIVE COMPOSITION

In previous work [9] we have defined a kernel language for the composition of invasive patterns, a simplified version of which, sufficient for illustration purposes here, is shown in Fig. 3. The kernel language is based on the following key concepts: composition programs ($Prog$) are sequences of invasive patterns $P$, each of which is defined as a pair of operators $O$. The first operator (possibly invasively) extracts information from a set of source computations or hosts; the second applies (possibly invasive) modifications to a set of targets. Finally, operators are defined as pairs of event sequences, the first of which defining the context where the operator is to be applied, the second defining an adaptation to be applied when the operator context matches the current execution context.

The pattern-based composition program represented in Fig. 2 can be expressed with our language via the composition of different patterns characterized by event sequences as follows:

$$
\begin{array}{l}
P := \langle farm_1, farm_2, gather_3, gather_4 \rangle \\
\text{where} \\
\quad \ldots \\
\quad farm_2 := (\ (e_1@l2,\ \epsilon),\ (\epsilon,\ \{e_{1,i}@l_i\}_{i\in\{1,2,3\}}\ ) \\
\quad gather_3 := (\ (\epsilon,\ \{e_{2,i}@l_i\}_{i\in\{1,2,3\}}),\ (\epsilon,\ e_3@l_2)\ ) \\
\quad \ldots \\
\text{and} \\
\quad e_0 = \ getEncounters(..)@l_0 \\
\quad e_1 = \ getEncounters(..)@l_i
\end{array}
$$

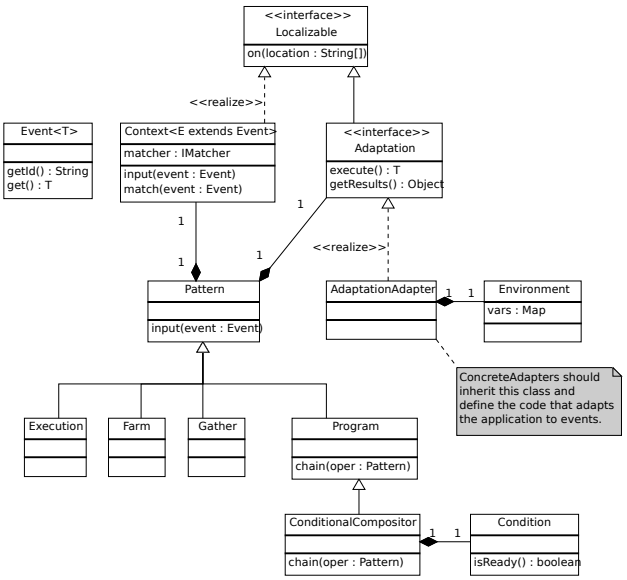


**Figure 4: Framework for invasive composition**

$$
\begin{array}{l}
e_2 = \ getSummarizedEncounter(..)@l_1 \\
\ldots \\
e_5 = \ summarizeEncounters(..)@l_0
\end{array}
$$

Here the composition program $P$ is defined using a sequence of four composition operators. The operator $gather_3$ is defined as an (outer) pair whose first component represents the source information to be gathered from the three nodes and the second pair the information to be integrated (invasively) in the target node. The sources and target are defined as (inner) pairs whose first components, both empty ($\epsilon$) here, represent context information used to be generated via the events that are transmitted and integrated by means of the second components of respectively the first and second inner pair. Events define interesting points in the application *e.g.,* $e_1$ define a context to obtain the information of the encounters of a patient, but we are interested in a summarized view represented by the new service invocation $e_2$ and finally consolidated in $e_5$.

We have extended our previous work, by designing and implementing a framework for Java for invasive composition. This framework directly represents the basic concepts (composition programs, invasive patterns and operators), while providing a more powerful composition model than represented by the simplified kernel language above.

Fig. 4 presents a high-level view of the main abstractions of our invasive operator library. Concretely we have modeled the main abstractions as interfaces that can be specialized. `Localizable` marks the elements which may belong to different locations. Class `Context` enables contexts to be defined by the matching of regular expressions of events (using a regular expression matcher implementing the `IMatcher` interface). `Event`s are implemented using a parametric container class. Programmers may tailor contexts to their needs; we provide, however, a base implementation that is sufficient to match regular expressions over call-like pointcuts. An `Adaptation` defines a sort of task execution with an environment to receive and return arguments. Programmers should spe-

```
1  public List<Encounter> getEncounters(
2    Patient who, Location loc, Date fromDate, Date toDate,
3    Collection<Form> enteredViaForms,
4    Collection<EncounterType> encounterTypes,
5    Collection<User> providers, boolean includeVoided) {
6
7    // Aspect Injected Before Code
8    Context context = new Context(this);
9    Adaptation adaptation =
10     new AdaptationAdapter("EncounterService.getEncounters",
11       who.getPatientId()) { ... };
12   Farm farm2 = new Farm(context, adaptation);
13   farm.input(this);
14   farm.execute();
15
16   // Traditional execution
17   List<Encounter> encounters =
18     dao.getEncounters(who, loc, fromDate, toDate,
19       enteredViaForms, encounterTypes, providers,
20       includeVoided);
21
22   // Injected Compositional Code
23   Gather gather3 = new PartialGather();
24   Composition.parallel(farm2, gather3);
25   SummarizedEncounter summarizedEncounter =
26     joinSummarizedInformatiaon(gather.getResponses());
27   encounters.add(summarizedEncounter);
28   return summarizedEncounters;
29 }
```

**Figure 5: Distributed encounter consolidation code**

| Class/Interface | Invasive Modifs | Description |
|---|---|---|
| EncounterService | 3 | Modify interface def. and two method impl. to prepare the new distributed request and reception of data |
| PatientService | 1 | Add collection for summarized encounter info for patients |
| ObsService | 1 | Limit observation info to relevant fields |
| OrderService | 2 | Add and manage order summaries |
| Forms | 4 | Handle new info to create records for distributed patient info |
| Reports/Views | 5 | Add distributed info as part of reports and views |

**Table 1: Modified elements to implement the distributed medical record**

cialize this class in order to add their own behavior if a given context is matched. Finally, an invasive `Pattern` is a composition of a context and an adaptation. The pattern class subsumes the operators presented in the kernel language, which provide simpler contexts and adaptations that consist solely of event sequences. The concrete implementation of the operators such as `Execution`, `Farm` and `Gather`, refine the semantics of the distributed characteristics of the given operations. And connectors such as: `Program` and `ConditionalCompositor` define the way patterns are composed: the execution of a successor pattern on a node is started as soon as the execution of the adaptation of a predecessor pattern has terminated.

# 4. OpenMRS EVOLUTION REVISITED

We have implemented the OpenMRS ConsolPatData evolution using the composition framework. Fig. 5 shows the inner part of the pattern composition. Here we show only the code of the inner composition ($farm_2 \rightarrow gather_3$) corresponding to the patterns and language that were presented previously in Fig. 2 and 3. The patterns involved in this composition are nested farms, as well as partial gathers which collect information by allowing for timeouts. The composition shows in line 24, in particular, the use of composition operators over primitive invasive patterns that have been constructed by defining appropriate contexts and adaptations, as exemplified for the primitive farm pattern in lines 8–13.

Table 1 summarizes the different modifications done in order to evolve the system to support the distributed medical examination (the table only includes non deprecated methods).

The implementation of the distributed medical record requirement has been achieved with the use of our Invasive Patterns library. It allows us to program directly the involved distribution patterns and their protocol (e.g. in the case of the partial gather with timeout). We have implemented the complete solution in a concise and relatively simple way that involves one class which defines the patterns and its interaction points, and two additional utility classes that deal with concrete details of the adaptations.

# 5. RELATED WORK

Evolution and integration of distributed systems are complex tasks that require careful planning and execution. Currently, engineers execute such tasks by means of careful application of design patterns and manual modification of interfaces and implementations. Patterns for distributed applications are typically only used as informal sets of best practices for the modification of communication and interconnection structures. Furthermore, the problem of adapting interfaces to make different systems and components compatible is ordinarily also addressed by means of manual manipulation of code. Not many of the existing works proposes a more structured approach to composition. Finally, the problem of evolution and integration of distributed systems is intimately linked with the evolution of individual components. Here again, manual manipulation of code is the preferred mechanism of evolution. However, recent work on the refactoring using transformations and tool support based on Model Driven Engineering are striving for a more structured approach. In the remainder of this section we discuss related work to each of the mentioned problems of evolution and compare them with our proposal.

A fair number of approaches to the refactoring of applications using transformations have been considered. A first set of approaches addresses the problem of defining general frameworks or metamodels [8, 12, 10] and allow common refactoring actions over different programming languages and models to be expressed. These approaches extract the commonalities of refactoring actions and allow developers to apply refactoring actions on their programs and models. Other approaches have proposed modeling and transformation of source code to address refactoring actions for specific languages [13, 3], or the manipulation of source code

by means of the specification of refactoring examples [5]. All these approaches address the problem of code restructuring but do not, in contrast to our approach, address the problem of refactoring complex communication patterns found in distributed systems. Furthermore, no support for complex pattern compositions is provided, as we do with our composition framework.

Concerning interface modifications, invasive Composition as proposed by Aßmann [1] adapts and extends components at hooks by means of transformations. In this approach modifications can be performed at explicit hooks that are provided by the component developer and implicit hooks as common abstractions provided by the programming platform, e.g. method entry points. Note that this approach is similar to ours in the sense that it admits invasive modification of composition interfaces, however, our approach goes a step further by allowing explicit manipulation of complex communication patterns. Furthermore, pattern compositions have to be implemented in an ad hoc manner.

Finally, other approaches use distribution patterns and parallel patterns for integration of distributed systems, and the configuration of parallel algorithms respectively [6]. Hoppe and Woolf proposed Enterprise Integration Patterns [7], a catalog of best practices to address common problems found during integration of distributed systems using messaging middleware. Their approach explicitly addresses the manipulation of asynchronous communication, and provides several patterns for the composition of program communication behavior. However, these techniques rely on ad hoc manipulation of applications involved in the integrated system. In the domain of massively parallel applications, patterns are used to compose parallel algorithms over homogeneous deployment environments [4] and do not support invasive composition.

# 6. CONCLUSION

In this paper we have motivated that evolution tasks in OpenMRS, a health information systems, require the invasive composition of interfaces and implementations. Concretely, we have shown that the consolidation of patient data from remote sites needs several interfaces and classes at different levels of the OpenMRS architecture to be modified. We have presented a new composition framework in Java that supports the expressive definition of pattern-based invasive compositions. Finally, we have provided evidence that our framework enables the evolution of patient data consolidation in a systematic manner and that pattern compositions significantly facilitate this evolution task.

# 7. REFERENCES


[1] Uwe Aßmann. *Invasive Software Composition.* Springer Verlag, 2003.

[2] Luis Daniel Benavides Navarro, Mario Südholt, Rémi Douence, and Jean-Marc Menaud. Invasive patterns for distributed applications. In *Proceedings of the 9th International Symposium on Distributed Objects, Middleware, and Applications (DOA'07)*, LNCS. Springer Verlag, November 2007.

[3] Enrico Biermann, Karsten Ehrig, Christian Köhler, Günter Kuhns, Gabriele Taentzer, and Eduard Weiss. Graphical definition of in-place transformations in the eclipse modeling framework. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 425–439. Springer Berlin / Heidelberg, 2006.

[4] S. Bromling, S. MacDonald, J. Anvik, J. Schaeffer, D. Szafron, and K. Tan. Pattern-based parallel programming. In *Proceedings of the 2002 International Conference on Parallel Processing (31th ICPP'02)*, Vancouver, Canada, August 2002. Univ. of Toronto.

[5] Petra Brosch, Philip Langer, Martina Seidl, Konrad Wieland, Manuel Wimmer, Gerti Kappel, Werner Retschitzegger, and Wieland Schwinger. An example is worth a thousand words: Composite operation modeling by-example. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, MODELS '09, pages 271–285, Berlin, Heidelberg, 2009. Springer-Verlag.

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

[7] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, building, and deploying messaging solutions.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.

[8] Ralf Lämmel. Towards generic refactoring. In *Proceedings of the 2002 ACM SIGPLAN workshop on Rule-based programming*, RULE '02, pages 15–28, New York, NY, USA, 2002. ACM.

[9] Ismael Mejía and Mario Südholt. Structured and flexible gray-box composition: application to task rescheduling for grid benchmarking. In *Proceedings of the IADIS International Conference on Applied Computing 2010*, Timisoara, Romania, October 2010.

[10] Naouel Moha, Vincent Mahé, Olivier Barais, and Jean-Marc Jézéquel. Generic model refactorings. In Andy Schürr and Bran Selic, editors, *Model Driven Engineering Languages and Systems*, volume 5795 of *Lecture Notes in Computer Science*, pages 628–643. Springer Berlin / Heidelberg, 2009.

[11] OpenMRS home page. http://openmrs.org.

[12] Jan Reimann, Mirko Seifert, and Uwe Aßmann. Role-based generic model refactoring. In Dorina Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Model Driven Engineering Languages and Systems*, volume 6395 of *Lecture Notes in Computer Science*, pages 78–92. Springer Berlin / Heidelberg, 2010.

[13] Gabriele Taentzer, Dirk Müller, and Tom Mens. Specifying domain-specific refactorings for andromda based on graph transformation. In Andy Schürr, Manfred Nagl, and Albert Zündorf, editors, *Applications of Graph Transformations with Industrial Relevance*, volume 5088 of *Lecture Notes in Computer Science*, pages 104–119. Springer Berlin / Heidelberg, 2008.

[14] WMP Van Der Aalst, AHM Ter Hofstede, B. Kiepuszewski, and AP Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.