

Second International Workshop on Variability & Composition (VariComp 2011)

In applying multi-dimensional separation of concerns, the composition of carefully separated concerns is an important issue. The time at which concern composition is applied can vary depending on the concrete approach at hand. Aspect-oriented techniques have extensively been used to enable variability in software product lines, where features are usually composed at build time. Likewise, run-time composition has been investigated, enabling dynamic aspect weaving, recomposition and reconfiguration. Using context-oriented programming, dynamic feature variation is used to react to environmental changes and events in a way statically controlled by the programming language. This workshop constitutes a forum for researchers working on tools and techniques to support the aforementioned composition stages, potential implementation and optimization approaches as well as formalization and verification techniques.

The objective of this workshop is to bring together researchers and practitioners concerned with software variability management and composition techniques, in particular those that improve modularity. The workshop is aimed at fostering cross-fertilization of a variety of areas, always with a view to addressing early and late software variability and composition.

The workshop goals in particular are:

- To serve as a forum for the discussion of emerging ideas in variability and composition.
- To help researchers find complementary views on their work (including feedback and related work).
- To foster new interactions and collaborations among researchers of different institutions.
- To serve as venue in which researchers meet physically, thus building a stronger professional bond than is possible with electronic communication.

We hope that the contributions for the workshop and the discussions during the workshop will help providing you interesting new insights, and that you take the opportunity to exchange and discuss ideas and experiences with other participants during the workshop.

The VariComp 2011 workshop organizers,

Walter Cazzola
Department of Informatics and
Communication
University of Milano
Italy

Sebastián González
ICTEAM|
UCLouvain
Belgium

Michael Haupt
Hasso Plattner Institute
University of Potsdam
Germany

Philippe Lahire
Laboratoire I3S
University of Nice-Sophia Antipolis
France

Stefan Van Baelen
DistriNet
K.U.Leuven
Belgium

Configuration Knowledge of Software Product Lines: A Comprehensibility Study

Elder Cirilo¹, Ingrid Nunes^{1,2}, Alessandro Garcia¹, Carlos J.P. de Lucena¹

¹Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Rio de Janeiro, Brazil

{ecirilo,ionunes,afgarcia,lucena}@inf.puc-rio.br

²King's College London, Strand, London, WC2R 2LS, United Kingdom

ABSTRACT

The configuration knowledge is a key element to the success of software product lines, as it defines constraints on how product line variability should be composed to derive products. Even though configuration knowledge specification is a long standing problem in software product line engineering, the impact of different specification techniques on comprehensibility has never been studied. This paper presents an empirical study to evaluate and compare three techniques for configuration knowledge specification. Each of them is centered on different means to express the configuration constraints: annotations, general-purpose modeling, and domain-specific modeling. Our results suggest that: (i) the use of domain-specific abstractions tends to facilitate the comprehension of coarse-grained variability; (ii) the use of general-purpose models imposes certain restrictions on the location and comprehension of the configuration knowledge; and (iii) the correct comprehension of configuration constraints is not associated with individual expertise.

Categories and Subject Descriptors

D.2.11 [Software Engineering]: Software Architectures—*Domain-specific architectures*; D.2.8 [Software Engineering]: Metrics

General Terms

Experimentation

Keywords

Software Product Lines, Configuration Knowledge, Controlled Experiment

1. INTRODUCTION

The adoption of configurable product lines for building families of enterprise systems is increasingly growing. A

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

VariComp '11 Porto de Galinhas, Pernambuco, BR

Copyright 2011 ACM 978-1-4503-0646-1/11/03 ...\$10.00.

configurable product line [4] is a subclass of software product lines that is tailored into single products without designing or programming new completion code. Systematic variability management as configuration potentially results in significant gains in software development such as reduced time-to-market and costs, but it requires explicitly specifying the configuration knowledge. This knowledge defines variability implementation and how it should be composed to derive products. If the activity of localizing and understanding the configuration knowledge requires a significant effort and costs to be performed, there is no payoff for the upfront investment made to build reusable software assets. Therefore, an easy-to-comprehend specification of all variability implementation and valid compositions is essential to the success of configurable product lines.

A particularity of enterprise product lines is the reuse of a large number of frameworks in concert for addressing a diversity of concerns (e.g., service composition and graphical user interfaces), mainly because they are ready-to-use infrastructures. As frameworks typically rely on a set of domain-specific abstractions, these become part of mindset of product line engineers along the software development. However, implementing variability as domain-specific abstractions involves making completion choices, some of which cut across heterogeneous languages (e.g., Java and XML). Since the completion code is not found in a single location, it tends to result in variability implementation spread across the product line architecture. As a consequence, it can lead to a negative impact on the activities of localizing and understanding the configuration knowledge specification.

A diversity of specification techniques were introduced as a way to facilitate the comprehension of the configuration knowledge [1, 2, 3]. Nevertheless, there are limited investigations about the implications of the adopted technique on the comprehensibility of framework-based software product lines. In fact, as far as we know, there is no study that analyzes the impact of specifying the configuration knowledge using the different available techniques, namely annotations, general-purpose modeling and domain-specific modeling. Existing research work [5] has analyzed and compared the individual techniques in terms of modularity, but they have not observed their level of impact on the configuration knowledge comprehensibility measures. In addition, their findings are not applicable to the context of enterprise product lines built with multiple frameworks, which is the scenario we are addressing.

We have performed an experimental study to evaluate and compare three tools that support the aforementioned tech-

niques: CIDE [3], which promotes colored-based annotations; pure::variants [1], which relies on general-purpose abstractions; and GenArch⁺ [2], which exploits domain-specific abstractions. In this study, six participants (individually organized) answered questionnaires about three framework-based product lines, each of which with the configuration knowledge modeled with the three different tools. The results indicate that:

- there is no relevant difference between the techniques when we consider the absolute number of correct answers. However, the time spent for answering them is significantly different;
- the use of domain-specific abstractions in the configuration knowledge specification, in fact, seems to facilitate the understanding of coarse-grained variability;
- because general-purpose modeling is abstract and hides many relevant details, it imposes certain restrictions to product line engineers to quickly localize and comprehend the configuration knowledge; and
- the correct comprehension is not associated with individual expertise about the frameworks that are part of the target product line.

The remainder of this paper is organized as follows. Section 2 describes the problem we are investigating and the evaluated approaches. Section 3 details how we have performed our experimental study, followed by Section 4, which presents results and associated discussion. Section 5 shows threats to the validity of our study. Finally, Section 6 concludes this paper.

2. CONFIGURATION KNOWLEDGE

The configuration knowledge expresses relations between product line variability and code assets, and their interactions. Several ways of specifying the configuration knowledge have been proposed [1, 2, 3]. This section overviews three different techniques selected for our study: annotation, general-purpose modeling and domain-specific modeling.

All investigated techniques assume that product lines are structured as a set of features. A feature model is used to represent the hierarchical arrangement of the features, together with the set of constraints that restricts their composition. Variability implementation is made using code configuration, i.e., the selection of specific parts of the code assets determines which parts are variable. These presence conditions are evaluated, in general, with respect to a feature model configuration, which is a subset of the available features. Therefore, given a valid feature model configuration, the evaluation of a configuration knowledge yields the code assets needed to build the corresponding product. Next we briefly describe what distinguishes each technique; further details can be found in their respective references.

Annotation. In annotative techniques, the configuration knowledge is specified directly into code assets similarly to `#ifdef` and `#endif` statements. Product line engineers annotate code fragments realizing variability inside the original source code. The annotations determine the presence of code fragments as part of the derived product. CIDE [3] is a representative tool that implements this technique. The advantage of CIDE in relation to other annotative approaches is its capability of warning product line engineers about annotations that do not respect the language syntax and type system. CIDE also provides views and navigation functionalities to support detailed understanding of the configuration

knowledge, as the ability of hiding specific variable parts.

General-purpose Modeling. This technique specifies the configuration knowledge in one or more general-purpose models. Restrictions are attached to model elements to define when an element must be included or excluded from the derived product. These types of restrictions define traces that connect features and configurations on code assets. Developers are also able to create *requires* and *excludes* constraints among model elements. Observe that these last types of constraints are difficult to be expressed through source code annotation. Pure::variants [1] is a commercial product line tool that represents this kind of technique. The configuration knowledge is specified in one or more object-oriented models, called family model. This tool also uses templates to support fine-grained variability.

Domain-specific Modeling. This technique allows product line engineers to use domain-specific abstractions in the specification of the configuration knowledge. As opposed to general-purpose techniques, this style of configuration knowledge specification does not allow product line engineers to freely create *requires* and *excludes* constraints among solution space elements, but these types of compositions must follow pre-defined specifications, for instance, the programming interfaces of frameworks. GenArch⁺ [2] offers an implementation of this technique. The configuration knowledge is specified through the use of three kinds of model: domain-specific knowledge models, implementation model and configuration model. Domain-specific knowledge models are expressed in terms of domain-specific abstractions, and model their composition rules. The implementation model represents the basic code assets (classes, aspects, files, folder, so on), and the configuration model captures the restrictions that constrain the inclusion of domain-specific and implementation model elements.

3. STUDY SETTINGS

The configuration knowledge is a key element for the success of configurable product lines and is essential for product line engineers to understand it with little effort. Therefore, our aim is to investigate whether the different techniques influence the correct comprehension of the configuration knowledge.

3.1 Experiment Hypotheses

The experiment aimed at verifying three inter-dependent hypotheses:

- H1:** *The correct comprehension of the configuration knowledge depends on the different specification techniques.*
- H2:** *The time to correctly comprehend the configuration knowledge depends on the different specification techniques.*
- H3:** *The individual differences among the expertise of product line engineers do not impact on the correct comprehension of the configuration knowledge.*

3.2 Selected Product Lines

As part of the experimental procedure, we have selected three different software product lines in order to perform our study, whose characteristics are summarized in Table 1. In this table, each of them is described by: its name and domain of application; the frameworks used to implement them; its size in terms of the number of features (mandatory, optional and alternative); and the most common type of

Name	Domain	Frameworks	Size	Variability Granularity	Variability Implementation Techniques
E-Shop	Online stores	SpringDM SpringMVC iBatis	18 features • 9 Mandatory • 7 Optional • 2 Alternative	Coarse-grained Fine-grained	Inheritance and polymorphism
OLIS	Personal assistance software	Struts 2.0 Spring Hibernate JADE Jadex	21 features • 13 Mandatory • 5 Optional • 3 Alternative	Coarse-grained	Design patterns Inheritance and polymorphism Modularization into capabilities
Buyer Agent	e-Commerce agents	Jadex	12 features • 4 Mandatory • 1 Optional • 7 Alternative	Fine-grained	Goal decomposition Modularization into plans and capabilities

Table 1: Main Characteristics of the Target Product Lines.

configuration granularity (coarse- or fine-grained). Coarse-grained variability is related to classes, components, aspects or self-contained domain-specific abstractions (e.g., beans, actions, agents and capability), and fine-grained variability is related to attributes, methods, code segments or domain-specific abstractions that are part of some other abstraction (e.g., bean variants, beliefs, goals and plans). Finally, we also indicate the techniques adopted to support variability within the software product line architecture.

These product lines were chosen for several reasons: (i) they were implemented by experienced developers, which adopted widely used software development practices, such as design patterns, and traditional architectures; (ii) they vary in size; (iii) they take advantage of several commonly used application frameworks; and (iv) they vary in granularity. The inclusion of different granularity was important to enable us to observe if and when there was any effect of using abstractions on the specification of configuration knowledge. Moreover, we have selected product lines developed in our laboratory, due to the availability of developers. They helped us to model the configuration knowledge of these product lines in each of the tools, in order to assure the correctness of the configuration knowledge, which is essential for our study.

3.3 Background of the Participants

This initial evaluation involved six post-graduate (MSc and PhD) students answering questions about the three previous presented software product lines. All participants have knowledge in software product line engineering and in the languages Java and XML, but they were not familiar with the evaluated approaches. Therefore, they were given a short 60-minute demonstration of pure::variants, CIDE, and GenArch⁺. In this training session, we demonstrated specific functionalities of the tools and examples of configuration knowledge specification. We used a different product line to avoid biasing the experiment results.

In addition to training the participants, we asked each one to fill in a background form after answering questionnaires. Our aim was to survey about the expertise of participants in the frameworks used to implement the product lines. The expertise is a value ranging from 1 to 5, where 1 means no expertise in a given framework and 5 means a high expertise. After that, we calculated the degree of expertise of each participant for each product line. The degree of expertise in a given product line is the average of the expertise of the participant in the frameworks used to implement that product line. Table 2 summarizes the background of the participants. Rows 4-10 in Table 2 indicate the expertise that participants

Framework	Participant					
	P1	P2	P3	P4	P5	P6
Spring	4	4	4	2	1	3
Struts	2	2	1	4	2	2
Spring MVC	4	4	4	5	3	5
Hibernate	1	1	1	1	4	1
iBatis	2	2	2	3	1	3
Spring DM	1	1	1	1	1	1
Jadex	2	1	1	3	1	1
Product Line						
eShop	1.5	1.25	1	2.25	2	1
OLIS	3	3	2.75	3.5	1.75	3.25
Buyer Agent	2	1	1	3	1	1

Table 2: Degree of Expertise of the participants

claimed to have about each technology used to implement the different product lines. Most of them claimed to have little knowledge about the development of agent-oriented software systems with Jadex. Additionally, all the participants have not previously worked with service-oriented development using Spring Dynamic Modules (SpringDM). However, in general all participants have at least basic skills in the relevant frameworks of the experiment. Rows 12-14 in Table 2 indicate the resulting degree of expertise of the participants. Overall, most of them have satisfactory expertise in the studied product lines.

3.4 Experimental Design

With the aid of a training section the participants had to answer three questionnaires, one for each product line. The questionnaires are composed of ten questions involving the configuration knowledge. Examples of questions are: “Which *abstraction(s)/code asset(s)* is(are) related to the feature X?;” and “How many *abstraction(s)/code asset(s)* is(are) mapped to the feature Y?” Two dimensions were evaluated in the experiment: (i) correctness; and (ii) time. Therefore, we have evaluated not only if the subjects were able to correctly comprehend the configuration knowledge but also how fast they got the information they needed.

We designed our experimental study with the Latin square in order to control the test of each tool with each participant. The Latin square design gave us a random allocation of the tools in such a way that each one is used once for each participant (row) and once for each product line (column). Therefore, the size of the Latin square is 3 x 3, in which the x-axis is the participants and the y-axis is the product lines. We have replicated the square once. Table 3 shows the configuration of the Latin square, presenting the allocation of participants, product lines and evaluated tools.

Participants	E-Shop	OLIS	Buyer
P1 and P4	G+	PV	C
P2 and P5	C	G+	PV
P3 and P6	PV	C	G+

Table 3: Latin Square Configuration

4. ANALYSIS AND DISCUSSION

This section presents the results of our experiment and discusses general observations. The analysis was decomposed into two categories regarding to: (i) correct answers and time; and (ii) expertise.

4.1 Correct Answers and Time Analysis

Figure 1 shows a chart that relates the number of correct answers and the evaluated techniques. Each bar in this chart indicates the number of questions correctly answered by the participants (y-axis) for each combination of product line and tool (x-axis). The contractions C, PV and G+ stand for CIDE, pure::variants and GenArch⁺, respectively.

Product Lines vs. Correct Answers. Overall, the Buyer agent presented a superior number of correct answers when compared with the OLIS and E-Shop. The Buyer agent is characterized to be a simple product line, which relies only on one framework. Therefore, it required small number of elements in the configuration knowledge specification. On the other hand, the OLIS and E-Shop are larger and more complex product lines when compared with the Buyer agent. The E-Shop, in particular, contains several fine-grained variability, and most of the features crosscut different code assets implementing the architectural layers of this product line. Therefore, the E-Shop configuration knowledge cannot be fully modularized only through the use of domain-specific abstractions. From this result, we concluded that the size (number of features and lines of code) and the number of fine-grained variability have negatively influenced the number of correct answers and consequently the comprehensibility of the configuration knowledge.

Techniques vs. Correct Answers. We observed that the correct comprehension of the configuration knowledge do not depend on the different specification natures of CIDE and pure::variants. For example, participants 2 and 5 achieved the lowest number of correct answers in the E-Shop product line. Based on this value we conclude that CIDE does not seem to contribute for the participants to correctly understand scattered configuration knowledge, as originally claimed by its authors [3]. On the other hand, the general-purpose modeling technique implemented by pure::variants seems to have helped participants 3 and 6 to better understand the E-Shop specification. The advantage of pure::variants is that it provides an overview of the coarse-grained variability through the family model and a detailed understanding of fine-grained variability via templates statements inside code assets. Pure::variants also provides searching mechanisms that were widely used by some participants. These mechanisms apparently helped participants to filter the configuration knowledge. However, the same result was not observed for the OLIS product line, as CIDE presented a higher number of correct answers than pure::variants for this product line. From these contrasting results, we conclude that there is no particularity in these tools that make them significantly better than other.

Nevertheless, we observed that, in general, the use of

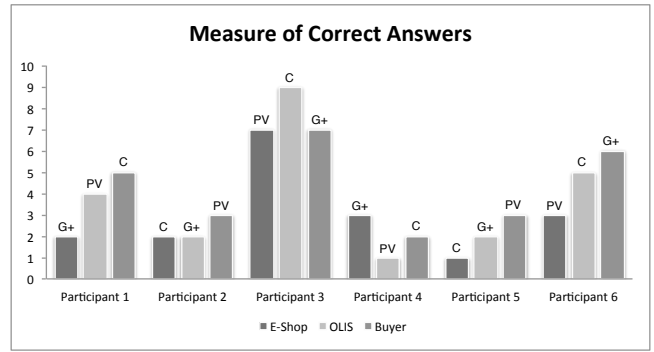


Figure 1: Measure of Correct Answers

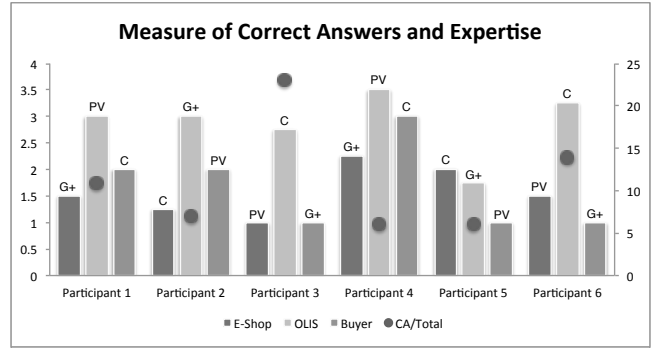


Figure 2: Measure of Correct Answers and Expertise

domain-specific abstractions provided the participants with the support to correctly understand the configuration knowledge. Participant 1 was the only exception, when pure::variants and CIDE were superior than GenArch⁺ in this aspect. To other participants, this tool presented intermediate results for the E-Shop and OLIS product lines and better results for the Buyer agent. Therefore, we conclude from this data that the hypothesis H1 holds. That is, product line engineers tend to better understand variability associated with abstractions in the presence of domain-specific modeling.

Approaches vs. Time. We also analyzed the time spent by the participants to localize and understand the configuration knowledge specifications. We observed that GenArch⁺ required the lowest time (3:40:45) for the participants to answer the questionnaires, followed by CIDE (4:19:56) and pure::variants (4:46:53). By analyzing only correct answers, we observed that GenArch⁺ was the tool that presented the lower average (0:02:57), and CIDE (0:03:10) and pure::variants (0:04:39) presented superior values.

Based on these results, we conclude that hypothesis H2 holds, but only with respect to some of the techniques. By comparing the time spent to answer questions correctly with GenArch⁺ and pure::variants, we observed that the use of the framework-provided abstractions improved the comprehension of the configuration knowledge specification, by allowing the participants localizing the configuration knowledge in a reduced time. However, there is no significant difference between GenArch⁺ and CIDE in terms of the time needed to localize and understand the configuration knowledge specification.

4.2 Expertise Analysis

Finally, we analyzed if the expertise of participant in the implementation frameworks was essential to correctly answer the questionnaires. Figure 2 shows the chart that relates the degree of expertise of each participant and his/her number of correct answers for each product line. Each bar in this chart indicates the expertise (x-axis) of participants (y-axis) about the product lines (bars). The bullets exhibit the total number of correct answers (CA/Total - secondary x-axis) of each participant. Note that there is no relation between the expertise and the number of correct answers. For example, participants 1, 2 and 3 claimed to have similar expertise; however, the participant 3 presented a superior number of correct answers. Correspondingly, participant 5, who has a limited expertise, presented a number of correct answers very close to the one achieved by participants 2 and 4, who claimed to have superior expertise.

We also compared the degree of expertise, number of correct answers and the product line tools. A high degree of expertise in the frameworks used to implement the OLIS product line combined with the annotative approach provided by the CIDE tool may have helped participants to correctly answer the questionnaire, however the same behavior was not observed in the other two product lines with the same tool. In contrast, the participants that use pure::variants to answer questions about the E-Shop product line presented a high number of correct answers despite they claimed to have a low degree of expertise. However, for the other two product lines the participants presented the same behavior described above, i.e. claimed to have a high degree of expertise but achieved a low number of correct answers. For GenArch⁺, we observed the same behavior described above: the expertise in the frameworks was not fundamental to correctly answer the questionnaire. As a result, we can conclude that there is no relation between the expertise and the number of correct answers, accepting the hypothesis H3.

5. THREATS TO VALIDITY

This section discusses the study constraints. For each category, we list possible threats and the procedure we took to alleviate their risk.

Conclusion Validity. The major external risk here is related to the engagement of the subjects to be part of the experiments, due to the length (time) of the questionnaires (almost two hours and thirty minutes for each participant). However, there was a rotation of the approaches order given that we adopted the Latin square. Another threat is the heterogeneity of participants. We have not taken any special care to select the participants and so they may represent random choices. Although the heterogeneity of subjects can also be considered a threat to the conclusion validity, it helps to promote the external validity of the study. Finally, the quality of the investigated tools is also a risk for the conclusion validity. However, we did not observe bugs that hampered the understanding of specifications or forced the participants to spend more time answering a question.

Construct Validity. We identified the following threats to the construct validity: confounding questions, and insufficient training session. To minimize these problems, we answered questions from participants as they were emerging. To avoid biasing the experiment results, we limited the explanations about tool functionalities to what was demon-

strated during the training session and about the questions to what clarifications were absolutely necessary.

Internal Validity. Threats to internal validity reside on how we have specified the configuration knowledge of the product lines with different techniques. We ensured that each product line has been specified following the same patterns in all tools by triple-checking each specification and by using the product line developers to model the configuration knowledge. It is important to be checked because the number of traceability links may increase depending on how the configuration knowledge is specified. In fact, the size and complexity of the product lines were two factors that have influenced the results.

External Validity. The major external risk here is related to the product lines. The selected product lines might not be representative of all industrial practices. To reduce this risk, we selected three product lines from different domains, which are heavily based on industry-strength frameworks. Although the size of the chosen product lines is limited, this decision allowed us to obtain more consistent results that could be interpreted in this specific context. Nevertheless, additional replications and statistical tests are necessary to determine if our findings can be generalized to other domains.

6. CONCLUSION

In this paper we presented an experimental study that compares three different product line tools (pure::variants, CIDE and GenArch⁺). The focus of our assessment is on the comprehensibility of configuration knowledge of framework-based software product lines. The experiment was performed based on questionnaires applied to six participants, following a Latin square configuration. As a result, GenArch⁺ tends to better support participants in the task of localizing and correctly answering questions about configuration knowledge. Therefore, based on current results, we can conclude that the correct understanding of the configuration knowledge is considerable dependent on the different specification techniques. Moreover, general-purpose modeling techniques might make it more difficult to product line engineers to quickly localize and comprehend the configuration knowledge. Interestingly, the individual expertise in the frameworks is not associated with the correct comprehension of the configuration knowledge. As future work, we intend to replicate the Latin Square in order to check whether the results found in this work prevails.

7. REFERENCES

- [1] D. Beuche. Modeling and building software product lines with pure::variants. In *SPLC '08*, page 358, USA, 2008. IEEE.
- [2] E. Cirilo and et al. Automating the product derivation process of multi-agent systems product lines. In *SBES '09*, pages 12–21, Brazil, 2009. IEEE.
- [3] C. Kästner and et al. Visualizing software product line variabilities in source code. In *SPLC '08*, 2008.
- [4] M. Raatikainen and et al. Characterizing configurable software product families and their derivation. *Software Process: Improvement and Practice*, 10(1), 2005.
- [5] M. Torres and et al. Assessment of product derivation tools in the evolution of software product lines: An empirical study. In *FOSD*, pages 1–8, 2010.

A study of 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 co-located parts of an application are to be executed in a remote location.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Varicomp '11 March 21-25, 2011, Pernambuco, Brazil
Copyright 2011 ACM 978-1-4503-0646-1/11/03 ...\$10.00.

- 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 system¹. 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

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

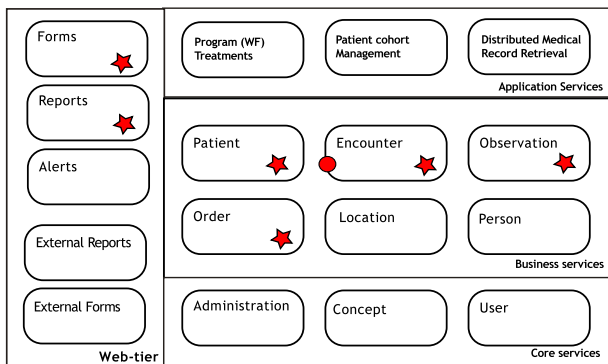


Figure 1: OpenMRS Architecture Diagram

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

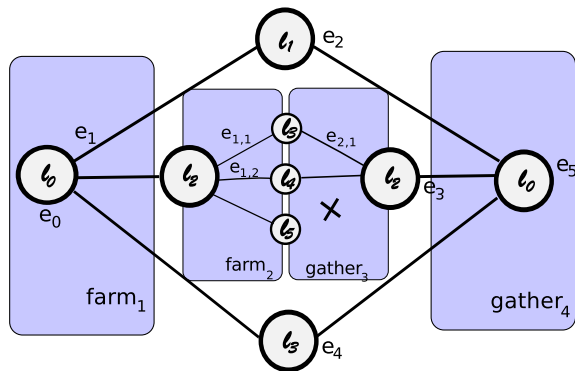


Figure 2: Pattern compositions for electronic medical record retrieval

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-

$$\begin{aligned}
Prog &::= \bar{P} && ; \text{ Programs} \\
P &::= (O, O) && ; \text{ Patterns} \\
O &::= (\bar{e}, \bar{e}) && ; \text{ Operators} \\
e &: (\text{Located}) \text{ Events}
\end{aligned}$$

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:

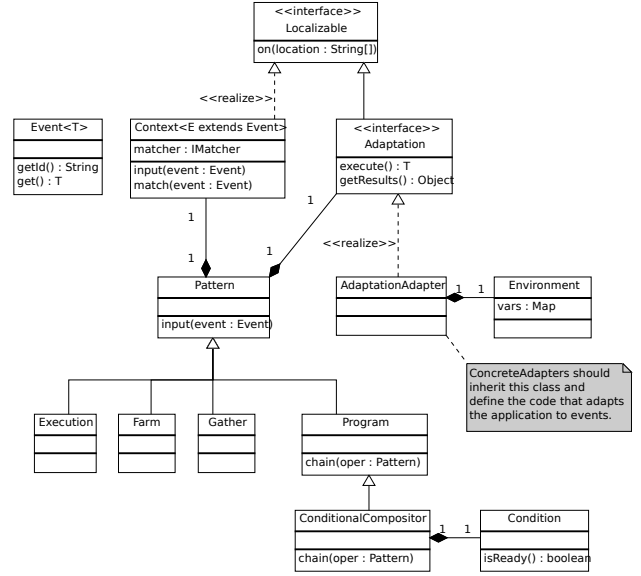
$$\begin{aligned}
P &:= \langle farm_1, farm_2, gather_3, gather_4 \rangle \\
\text{where} & \\
&\dots \\
farm_2 &:= ((e_1 @ l_2, \epsilon), (\epsilon, \{e_{1,i} @ l_i\}_{i \in \{1,2,3\}})) \\
gather_3 &:= ((\epsilon, \{e_{2,i} @ l_i\}_{i \in \{1,2,3\}}), (\epsilon, e_3 @ l_2)) \\
&\dots \\
\text{and}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


Figure 4: Framework for invasive composition

$$\begin{aligned}
e_2 &= getSummarizedEncounter(..) @ l_1 \\
&\dots \\
e_5 &= summarizeEncounters(..) @ l_0
\end{aligned}$$

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 (ϵ) 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). **Events** 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.getId(), loc, fromDate, toDate,
12 enteredViaForms, encounterTypes, providers,
13 includeVoided);
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 joinSummarizedInformation(gather.getResponses());
27 encounters.add(summarizedEncounter);
28 return encounters;
29 }

```

Figure 5: Distributed encounter consolidation code

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

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

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.