Software Metrics by Architectural Pattern Mining

Jukka Paakki[1], *Anssi Karhinen*[2],
Juha Gustafsson[1], *Lilli Nenonen*[1], *A. Inkeri Verkamo*[1]
[1]Dept. of Computer Science, PO Box 26, 00014 University of Helsinki, Finland
[2]Nokia Research Center, PO Box 407, 00045 Nokia Group, Finland
Email: {jukka.paakki,juha.gustafsson,lilli.nenonen,inkeri.verkamo}@cs.helsinki.fi;
anssi.karhinen@nokia.com
Fax: +358 9 19144441 (UH), +358 9 43766227 (NRC)

Abstract
A software architecture is the key artifact in software design, describing the main elements of a software system and their interrelationships. We present a method for automatically analyzing the quality of an architecture by searching for architectural and design patterns from it. In addition to approximating the quality of the design, the extracted patterns can also be used for predicting the quality of the actual system. The method is demonstrated by an industrial case over a complex telephone exchange software.

Keywords
Software metrics, software architecture, design patterns, anti-patterns, constraint satisfaction

1. Introduction

A generally accepted principle in software engineering is that the quality of a software system should be assured in the early phases of its life cycle. Quality assurance methods are most effective when capturing the requirements of the system and least effective when the system is already in full operational use: correcting an error encountered by an end-user is an order of magnitude more expensive than when finding it already in the requirements analysis phase.

Unfortunately, quality assurance methods that are applicable in the early development phases (such as reviews and inspections) rely on slow manual reading of documents with marginal possibilities for automation. On the other hand, methods with extensive automated support (such as testing) only apply in phases that are too late in the development life cycle to be really cost-effective.

The *software design* phase, acting as a bridge between informal, highly subjective user requirements and formal, precise implementation is a most natural place for effective quality assurance. On one hand, software design is one of the phases in software development where one still works on an abstract level without too much detail. On the other hand, the software design can be described in a precise manner using notations that can be automatically analyzed.

The main technical result of the design phase is a *software architecture*, describing the functional elements of the system and their interrelationships. In recent years, software architectures have been steadily growing in importance as one of the cornerstones of software quality, documentation, and reuse.

In this paper we present a technique for predicting the quality of a software system by its architectural analysis. Our technique is based on automatically finding *architectural patterns* whose properties are well-known to the extent that their effect on the behavior of the resulting system can be quantitatively measured.

We start by introducing the concept of software architecture in Section 2 and the applied pattern classes in Section 3. The pattern mining algorithm and a tool based on it are presented in Section 4, followed by a summary of software metrics supported by the tool in Section 5. An industrial case is discussed in Section 6. Finally, conclusions are drawn in Section 7.

2. Software Architectures

The concept of “software architecture” has been under intensive research during the last few years. The area is still somewhat immature, and no single standard definition for the term exists. Usually a software architecture is understood as an abstract structural

description of the software system in terms of its main components and the relationships among them [1]. A software architecture is a technical artifact, meaning that it is given in a particular (semi-)formal notation so as to be analyzable.

A software architecture is usually a rather complex entity. To manage the complexity, it must be possible to divide the architecture description into smaller units. Usually these units stand for different architectural *views* that complement each other.

Several view-based architectural models have been developed. One of the most popular ones is the "4+1 model" which organizes a software architecture into the following views [2]:

1. The *logical view* describes the static structure of the system, as derived from its domain.
2. The *process view* describes the (dynamic) concurrency, distribution and synchronization aspects of the system.
3. The *development view* shows the (static) organization of the system in terms of technical facilities of the development environment.
4. The *physical view* describes the mapping of the system onto hardware, databases, and communication infrastructure.
5. The *scenarios* tie the other views together into externally usable system services.

To describe the architecture of the software system comprehensively, a language is needed that supports all the central viewpoints. A real architecture description language must provide description facilities at the architectural level (that is, as "components" and "connectors") and it must have precise syntax and semantics. To support human intercourse, the language should also be visual.

While expressive dedicated architecture description languages have been developed, they are not very popular. Instead, it seems that the general-purpose object-oriented and visual modeling language *UML* (Unified Modeling Language) [3] will take the role of a leading architecture description language.

Even though UML does not support the description of software architectures per se, its rich selection of extensible diagram types makes it expressive enough even in this area. Most notably, class diagrams, interaction diagrams, component diagrams, deployment diagrams, and use case diagrams of UML can be used for representing the logical view, the process view, the development view, the physical view, and the scenarios of the 4+1 model, respectively. We also have chosen UML as our core description language, mainly due to its industrial relevance.

3. Architectural Patterns

The area of software design has a relatively long history with a number of sound technical achievements such as the principles of modularization, information hiding, coupling and cohesion. These have mainly existed as informal guidelines whose realization each software designer or architect has to create by herself, in different forms in different situations.

Especially the object-oriented software community has recently tackled this ad-hoc state of the art by collecting and documenting widely used solutions to frequently occurring software problems. The most well-known class of these are *design patterns* [4] that describe reusable and extensible technical solutions to common design problems in a standard object-oriented format. Other forms of the same concept are, e.g, higher-level *architectural patterns* and lower-level programming *idioms* [5].

The general idea behind these patterns is to describe a sound design solution in an explicit (object-oriented) form that applies in a certain context, and to analyze the benefits and drawbacks of the solution. Usually the patterns have been found from existing systems, but they can also be created from scratch.

What makes patterns a powerful design tool is the fact that they can be embedded as such or with small modifications into a software architecture. Since the documented patterns have been found highly mature, they implicitly bring a significant amount of discipline and quality into software design. Furthermore, the documented properties, merits and drawbacks of design patterns make it possible to predict the quality of a software system from its architecture that contains instances of patterns.

As an example, Figure 1 shows the structure of the *Abstract Factory* design pattern [4] as a class diagram. The pattern provides an object-oriented interface (role *AbstractFactory* in the pattern) for operations (*Creates*) that create families of related objects (*Abstracts*) without specifying their concrete classes (*Products*).

In addition to "good" (design) patterns, also "bad" software patterns have been recognized. Such a pattern, when applied in software development, sooner or later leads into some kind of trouble. One set of "bad" patterns has been documented under the term *AntiPattern*: an AntiPattern is a literary form that

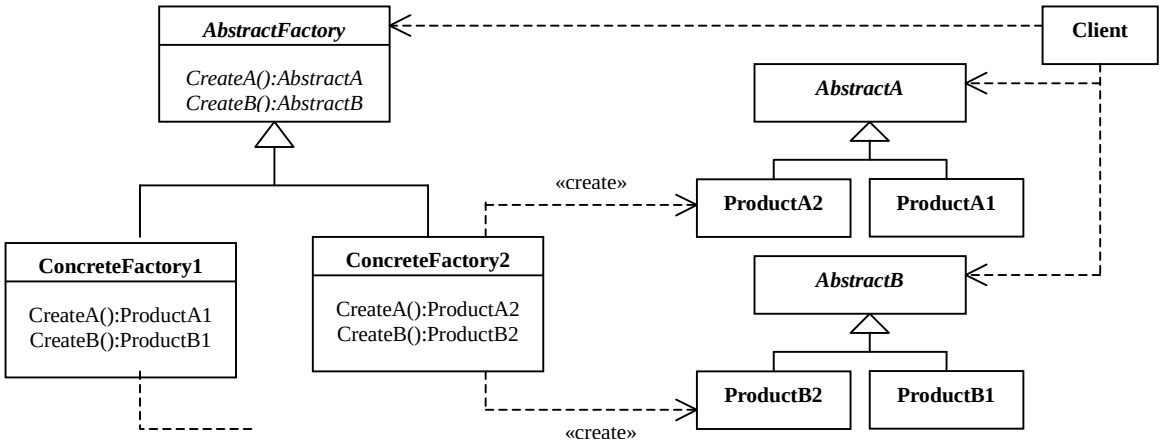

Figure 1. *Abstract Factory* design pattern.

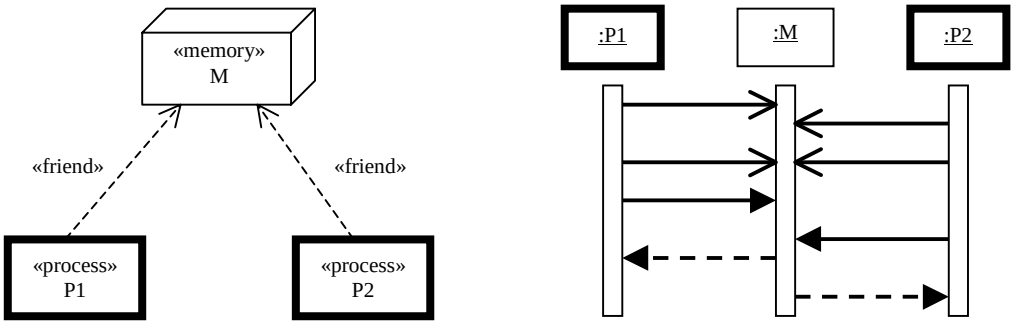

Figure 2. *Common Coupling* anti-pattern.

describes a commonly occurring solution to a problem that generates decidedly negative consequences [6]. In the following we will use the general term *anti-pattern* for such "bad" patterns.

Anti-patterns may be technical, or more related to general software processes and projects. When detecting them in order to uncover potential quality problems in a software architecture, just the technical class of anti-patterns is applicable. Finding an instance of an anti-pattern from an architecture typically means that the architecture should be improved by refactoring the suspicious instance into a more robust form – e.g, into an instance of a design pattern.

As an example, Figure 2 depicts the anti-pattern commonly known as *Common Coupling* between software modules [7]. In concurrent systems the pattern is known as “shared memory”, and in [6] as the AntiPattern “Blob”. In this case the anti-pattern is described by two UML diagrams, a deployment diagram on the left and a sequence diagram on the right. These diagrams emphasize different aspects of the pattern and thus complement each other in architectural analysis.

The deployment diagram for the pattern shows that there are two processes, *P1* and *P2*, accessing the same memory node. What makes this solution an anti-pattern is the fact that both processes have a *friend* dependency to the memory. Now the processes can freely access the shared information, violating the disciplined principles of interfacing and information hiding and making data corruption in the memory a potential danger.

The sequence diagram, on the other hand, shows that the processes access the shared memory by message passing. The upper half of the diagram specifies synchronous communication and potential concurrency (indicated by two messages at the same horizontal level). The lower half represents a sequence of procedure calls (arrows with solid head) and associated returns (dashed arrows), showing that the calls and returns for the processes *P1* and *P2* are interleaved and out of synchronism. Both these cases,

concurrent access and unbalanced interleaving, may be indications of mutual exclusion and synchronization problems in the system.

4. Pattern Mining in Maisa

Maisa (Metrics for Analysis and Improvement of Software Architectures) is a currently ongoing research project at the Department of Computer Science, University of Helsinki. The project develops methods and tools for automatically analyzing the quality of a software architecture and for predicting some central properties of the system founded upon the architecture. The project is financed by the Finnish National Technology Agency (Tekes), Kone, Nokia Mobile Phones, Nokia Research Center, and Space Systems Finland.

In Maisa, architecture analysis and quality prediction is based on the discovery of patterns from the description of the architecture, given as UML diagrams. There are two basic classes of patterns subject to architectural mining: "good" design patterns and "bad" anti-patterns. Instances of design patterns usually show robustness and discipline in design, whereas instances of anti-patterns might indicate poor solutions and lead to quality problems in the long run.

In our project pattern mining is considered as a *constraint satisfaction problem* (CSP). CSP has its roots in artificial intelligence where a large number of central problems can be formulated as a set of predicates (constraints) over variables in a particular domain [8]. Constraint satisfaction algorithms have been applied, for instance, in machine vision, scheduling, and program understanding.

More precisely, a CSP consists of a set V of variables, a finite and discrete domain D_i for each variable $i \in V$, and a set of constraints P (unary constraints P_i for single variables i, and binary constraints P_{ij} for pairs of variables i, j). The goal is to find an assignment S to the variables such that it satisfies all the constraints:

$$S = \{i := x \mid p_1 \wedge p_2 \wedge \ldots \wedge p_n \wedge p_{1j} \wedge p_{2j} \wedge \ldots \wedge p_{mj} \\ \forall\, p_k \in P_i\ \forall p_{kj} \in P_{ij}\ \forall i, j \in V, x \in D_i\}.$$

A number of constraint satisfaction algorithms have been developed [9]. In Maisa, we apply the arc consistency algorithm *AC-3* [8], where a CSP is represented as a graph in which the nodes stand for variables V with their domains D and the arcs stand for constraints P. The algorithm iteratively deletes from the node-domains such values that cannot satisfy the connected arc-constraints, until a solution (if any) is found. Since we want to find all potential pattern instances and have them verified by the user, our AC-3 implementation is exhaustive and interactive.

The pattern mining algorithm needs a software architecture and a set of subject patterns as input. The patterns and their metrics prediction attributes reside in a library and the user may choose which patterns she is interested in. Then the system searches for instances of the selected patterns in the architecture description and provides each match as a potential candidate to the user.

Since the constraints of a CSP can be conveniently expressed in first-order predicate logic, we have chosen (extended) Prolog as the structural coding of software architectures and architectural patterns. A similar solution has been applied when developing a tool that automatically recognizes design patterns in object-oriented (Java) programs [10]. Note that while Prolog is used for the representation of architectures and patterns, the actual implementation language of Maisa is Java.

The main components of an architecture / pattern and their relationships are expressed as Prolog facts. For example, the *Abstract Factory* pattern described in Figure 1 can be represented as follows:

```
class("AbstractA").
abstract("AbstractA").
class("AbstractFactory").
abstract("AbstractFactory").
class("ProductA2").
class("ConcreteFactory2").
inherits("ProductA2","AbstractA").
inherits("ConcreteFactory2","AbstractFactory").
has("AbstractFactory","CreateA()").
abstract("AbstractFactory.CreateA()").
returns("AbstractFactory.CreateA()", "AbstractA").
creates("ConcreteFactory2.CreateA()",
    "ProductA2").
returns("ConcreteFactory2.CreateA()",
    "ProductA2").
not same("AbstractFactory","AbstractA").
```

This description captures the relevant class hierarchies for product factories (with *AbstractFactory* as root) and for products (with *AbstractA*, *AbstractB* as roots). It also specifies that the final products shall be created by the concrete *CreateA*/*CreateB* methods. The last fact states that the class hierarchies for factories and products shall be distinct, so as to guarantee the central aspects of extensibility and reconfigurability of the pattern. The description could be extended, e.g.,

by stating that the *CreateA/CreateB* methods shall follow the *Factory Method* design pattern [4], as is done in [10].

As noted above, the AC-3 algorithm iteratively traverses the CSP as a graph. In this case the variables residing at the nodes of the graph stand for the classes and methods (e.g., *AbstractFactory*, *CreateA*) of the pattern and the arcs stand for the different kinds of relations between them (e.g., generalization: *inherits* and dependency: *creates*). The functor names used in the Prolog representation (e.g., class, inherits, creates) are taken from the standard UML meta model.

The worst-case complexity of the constraint satisfaction algorithm AC-3 is $O(ed^3)$ where e is the number of arcs in the constraint graph, and d is the domain size for the variables [9]. According to the tests reported in [10], the algorithm seems to be fast enough in practice: finding all the potential instances of the standard design patterns [4] from medium-size Java programs took just a few minutes. On the other hand, exhaustively mining a 100 KLOC program with 667 pattern matches took almost 20 hours.

In addition to [10], techniques for automatically discovering design patterns have been published, e.g., in [11, 12]. A general observation is that design patterns with an explicit structural form (such as *Abstract Factory*) can be recognized quite precisely, whereas patterns with loose structure and strong semantics (such as *Interpreter* [4]) are no subject for automatic search.

5. Software Metrics in Maisa

There exists a variety of software metrics that characterize properties of a software system [13] or an object-oriented design [14]. Most of these are implementation-level metrics that are computed from program code, and therefore appear usually too late in software development to be really useful. On the other hand, these estimates are quite accurate since they are generated from a precise description of the system.

The intended scope of the metric tool Maisa is the design phase of a system. The metrics supported by Maisa can be roughly divided into two categories [15]: (1) measures over the system architecture and (2) estimates over the final system. The architectural measures mostly address size or complexity in the style of traditional software metrics, examples being *number of classes*, *number of messages*, and *depth of inheritance hierarchy*. Notice that these measures are accurate and can be given as exact numbers.

The estimates, on the other hand, aim at predicting the quality of the actual system based on the analyzed design. The prediction is founded on the pattern mining approach described in the previous section. In the Maisa project, *size*, *performance*, and *complexity* (understandability) have been selected as the quality attributes to be foremost supported.

As was motivated when discussing the *Common Coupling* anti-pattern in Section 3, the patterns to be searched by Maisa can be specified in terms of several diagram types of UML. Usually both software architectures and patterns are specified for Maisa as class diagrams, activity diagrams, or sequence diagrams (or as a combination of these).

One problem to be studied is the mutual effect interacting, overlapping, and even conflicting patterns may have on the measures or predictions. For instance, what should be concluded if there is a significant overlap between a "good" design pattern instance and a "bad" anti-pattern instance in the architecture?

The main components of Maisa are a UML editor ("front-end"), a pattern library, a pattern miner, a metric analyzer, and a reporting tool. As mentioned in Section 4, Prolog is used as the internal representation format of architectures and patterns. Prolog is also used as the intermediate format between the external front-end and the actual Maisa tool set. In principle, any UML editor capable of exporting the diagrams in Prolog format can be used as the front-end; in the project we have selected a commercial customizable CASE tool as the default editor.

6. Case Study: Nokia's DX200 Switching System

Call control is one of the central functions of a telephone exchange. Call control functionality can be divided into different abstraction layers: The highest layer manages both a generic call between two subscribers, generally known as A- and B-subscribers, as well as all activities, like charging, that belong to generic calls. A layer below call control takes care of setting up the calls and tearing them apart. This is called the signaling layer.

A problem of call control and signaling in existing telecommunications networks is the great variety of different standards and conventions. Networks are usually connected together to form large interoperable telecom networks. Eventually all the public telecom networks of the world are connected. The users of these networks must be able to make

calls to all other users in other networks. This creates the typical problem of designing an abstract virtual system capable of dynamically supporting many different implementations.

Inside the telephone switches that form the core logic platform of the network, Nokia has implemented a generic model to support different call types through the network. The model is rich enough to support all native call types in different customer areas. There are abstract definitions with the required interfaces and functionality for all components needed to connect calls in this abstract layer. When a call is constructed (call-setup) these components must be instantiated and bound to actual implementations of signaling protocols and call management. In the object-oriented design paradigm this problem is known as a "virtual constructor".

To achieve correct instantiation of the components, the conditions that control the selection of correct implementations must be known. Of course this knowledge could be coded into the client that actually constructs the objects but conceptually that seems not to be the right place for such information. The abstract call components model the general functionality of call control in the network, not the relations and compatibilities between some signaling and call control protocols. The solution has shown to be robust, even though originally nobody knew that it actually is quite a common "pattern".

When the initial architecture of Nokia's DX200 telecom switches was carved, object-oriented design was not common activity, let alone rigorous usage of design patterns. Still the designs that were landed on some 15 years ago seem to hold well even in today's fast changing environment of telecom and data networks.

The basic idiom for DX200 is a communicating finite state machine, a paradigm that is directly supported by the definition and programming language SDL. In fact most important parts of the system have been implemented in an executable dialect of SDL. The basic implementation entity is a process family that can be mapped to several object-oriented design patterns. The process family models a service that can contain many types of subservices or functionalities. The master process receives service requests from the clients and depending on the type of the request activates a hand process of suitable type to handle the request and to take care of any further communication with the client.

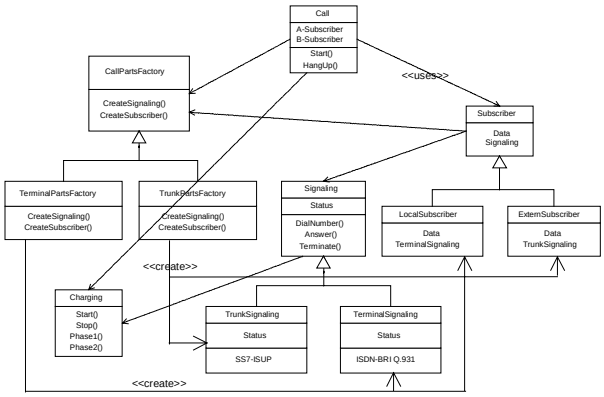

Figure 3. Call-control software architecture.

The usage of process families can be presented as a class diagram shown in Figure 3. When translated into Prolog format, the following facts are obtained. Here we omit most of the facts that are irrelevant for our case:

```
class(“Signaling”).
abstract(“Signaling”).
class(“CallPartsFactory”).
abstract("CallPartsFactory”).
class(“TrunkSignaling”).
class(“TerminalSignaling”).
class(“TerminalPartsFactory”).
class(“TrunkPartsFactory”).
inherits(“TrunkSignaling”,”Signaling”).
inherits(“TerminalSignaling”,”Signaling”).
inherits(“TerminalPartsFactory”,
    ”CallPartsFactory”).
inherits(“TrunkPartsFactory”,
    ”CallPartsFactory”).
has(“CallPartsFactory”,”CreateSignaling()”).
has(“CallPartsFactory”,”CreateSubscriber()”).
abstract(“CallPartsFactory.
    CreateSignaling()”).
abstract(“CallPartsFactory.CreateSubscriber()”).
returns(“CallPartsFactory.CreateSignaling()”,
    “Signaling”).
returns(“CallPartsFactory.CreateSubscriber()”,
    “Subscriber”).
creates(“TrunkPartsFactory.
    CreateSignaling()”,“TrunkSignaling”).
returns(“TrunkPartsFactory.
    CreateSignaling()”,“TrunkSignaling”).
not same(“CallPartsFactory”,”Signaling”).
class("Call").                    …
class("Subscriber").
uses(“Call”,”Subscriber”).        …
class("LocalSubscriber").         …
```

```
class("ExternSubscriber"). ...
class("Charging").         ...
```

By mapping this architecture description with pattern descriptions such as those given in Section 4, our pattern mining tool Maisa can isolate architectural patterns in the diagram. With the tool we can extract, for example, the *Abstract Factory* design pattern in action. Actually we find two instances of the basic pattern: one with the abstract call component (*Call* in Figure 3) as client and the other with the subscriber component (*Subscriber*) as client. Both instances share the same factory components.

Which call control and signaling components should be instantiated depends on the topology of the network, the physical connections of the A- and B-subscribers of the call, and possibly some national or operator specific variations and special rules. All this decision logic is abstracted into the factory components.

The CSP algorithm returns, among others, the following bindings that span one of the *Abstract Factory* instances. The facts applied for generating these bindings are emphasized in the list above.

```
AbstractFactory    := CallPartsFactory
AbstractA          := Signaling
ProductA2          := TrunkSignaling
ConcreteFactory2 := TrunkPartsFactory
CreateA            := CreateSignaling
```

The found pattern instances can be used to analyze the architecture and to predict future evolution of the system, based on the information stored in Maisa's pattern library. For instance, in this *Abstract Factory* case one could conclude that the exchange of process families is easy and consistency among signaling and subscribing elements is promoted, but supporting completely new kinds of product elements is difficult [4]. Also quantitative prediction can be made, for instance by reporting that one instance of this particular pattern typically creates 100 lines of code into the corresponding software.

The detection of such patterns from legacy systems with Maisa requires representing the design in a standard notation (SDL in this example case) and coding it into Prolog format. The construction of such an architectural representation is usually too laborious to be done manually, but fortunately the construction can be automated with reverse-engineering tools [16].

Such a reverse-engineering tool for the recovery of architectural information from software code is currently under development in another research project, *Saara* (Software Architecture Analysis, Recovery and Assessment). Architectural quality of legacy systems, such as DX200, can then be measured by first extracting their architecture by Saara and then analyzing the architecture and its patterns by Maisa.

As previously noted, not all the design patterns are absolutely "good". Especially old legacy systems contain examples of design patterns that have evolved into anti-patterns. These are design patterns that are systematically used but that have lost their original justification in architectural decay.

An example of such a pattern in a distributed system such as DX200 is the usage of shared memory to communicate between processes (see the pattern in Figure 2). Typically this is used in the first place to circumvent any inefficiency in the message passing mechanism inside one physical computer node. In the long run the pattern fires back as it makes it impossible to allocate tightly-coupled processes with shared memory into different physical computers. Thus the pattern that was approved to improve performance can eventually prohibit the scalability of the system into new levels of performance.

Notice that while this particular case is based on the analysis of an existing legacy system (which also is important for future development and maintenance), the intended application area of Maisa is architectural analysis at the design phase of the system. Using Maisa during design makes it possible to localize design problems before implementing them in the code. For instance, the potential problem of using shared memory could have been noticed already at the design phase when analyzing its architecture with Maisa.

7. Conclusions

Architectural pattern mining offers a practical way to evaluate the quality of a software system already at the design phase, by tracking down and analyzing patterns that have been introduced into the architecture of the system. This makes it possible both to validate the quality of the design in terms of design patterns and to recognize potential problems as anti-patterns. In the latter case the analysis can direct rearchitecting and evolution of system design.

We have presented an approach to architectural pattern mining. In our method the subject architecture and the relevant patterns are described as UML diagrams which are translated into Prolog format. This intermediate format has been chosen because it makes it possible to treat pattern mining as a constraint

satisfaction problem.

The method is being developed in the Maisa project whose main objective is to predict the system's quality attributes such as size, performance, and complexity on the basis of patterns that are found in the software architecture. Experiences with a related reverse-engineering tool that discovers design patterns from program code demonstrate the potential practicality of the approach: precisely described patterns can be automatically found from a software architecture reasonably fast and with a small fraction of misses or false positives [10].

What is unique in our approach is the combination of pattern discovery and software metrics for evaluating a software architecture, and the use of the generated information in predicting the quality of the system under development. In general, automated quality assurance of software architectures is a rather unexplored area. Another novel technique in Maisa is the recognition of *anti*-patterns and the estimation of their effects on software quality.

Pattern discovery is not completely precise. Since a pattern is always expressed in an abstract generic form, it also involves some degree of inaccuracy. In Maisa we try to avoid this problem by automating the discovery of those patterns only whose structure is precise enough, and by letting the user verify the candidates. Moreover, patterns are not the only source for metrics computation but we also apply conventional software metrics induced over the complete architecture. Notice that since the patterns and the subject architecture are expressed at the same conceptual level (as UML diagrams), the well-known problems of conceptual mismatch in design recovery from software code can be avoided.

The Maisa tool is currently under development. In addition to architectural pattern mining, research effort is concentrated on developing a UML-based diagrammatic formalism for describing and analyzing real-time and performance aspects of a software system [17]. Note that standard UML falls short in this respect, so something more expressive is needed (see, e.g., [18, 19]).

REFERENCES

1. L. Bass, P. Clements, R. Kazman: *Software Architecture in Practice*. Addison-Wesley, 1998.
2. P.B. Kruchten: The 4+1 View Model of Architecture. *IEEE Software* 12, 6, 1995, 42-50.
3. J. Rumbaugh, I. Jacobson, G. Booch: *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.
4. E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
5. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal: *Pattern-Oriented Software Architecture – A System of Patterns.* John Wiley & Sons, 1996.
6. W.J. Brown, R.C. Malveau, H.W. McCormick III, T.J. Mowbray: *AntiPatterns – Refactoring Software, Architectures, and Projects in Crisis.* John Wiley & Sons, 1998.
7. R.S. Pressman: *Software Engineering – A Practitioner's Approach*. McGraw-Hill, 1997.
8. A.K. Mackworth: Consistency in Networks of Relations. *Artificial Intelligence* 8, 1, 1977, 99-118.
9. V. Kumar: Algorithms for Constraint-Satisfaction Problems: A Survey. *AI Magazine* 13, 1, 1992, 32-44.
10. P. Misikangas: Recognizing Design Patterns from Object-Oriented Programs (in Finnish). Report C-1998-1, Department of Computer Science, University of Helsinki, 1998.
11. C. Krämer, L. Prechelt: Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. In: *Proc. Working Conference on Reverse Engineering*, Monterey, 1996. IEEE CS Press, 1996, 208-215.
12. G. Antoniol, R. Fiutem, L. Cristoforetti: Using Metrics to Identify Design Patterns in Object-Oriented Software. In: *Proc. 5th Int. Software Metrics Symposium*, Bethesda, ML, 1998. IEEE CS Press, 1998, 23-34.
13. N. Fenton: *Software Metrics*: *A Rigorous Approach*. Chapman-Hall, 1991.
14. S.R. Chidamber, C.F. Kemerer: A Metrics Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering* 20, 6, 1994, 476-493.
15. L. Nenonen, J. Gustafsson, J. Paakki, A.I. Verkamo: Measuring Object-Oriented Software Architectures from UML Diagrams. Submitted for publication.
16. The *FAMOOS* project: http://www.iam.unibe.ch/~famoos.
17. J. Gustafsson, L. Nenonen, J. Paakki, A.I. Verkamo: Performance Modeling in UML. Submitted for publication.
18. C. Smith: *Performance Engineering of Software Systems*. Addison-Wesley, 1990.
19. B.P. Douglass: *Real-Time UML*: *Developing Efficient Objects for Embedded Systems*. Addison-Wesley, 1998.