FEATURE MODELS, PATTERN LANGUAGES AND SOFTWARE PATTERNS: TOWARDS A UNIFIED APPROACH

Markku Hakala
Tampere University of Technology
markku.hakala@cs.tut.fi

1. INTRODUCTION

Feature models [Kan90] and pattern languages [Ale79] are similar ways of specifying domain specific languages (DSL). Whereas feature models may be used in deriving a matrix computation library from the family of matrix computation libraries [CzE00], a pattern language could be used in deriving a shopping mall from a family of shopping malls [Ale75]. Both approaches are ways of specifying a domain as a language, so that given the requirements of a specific application it is possible to derive a sentence of that language which leads to a concrete solution.

Fred [Hak01a, Hak01b] is a programming environment that allows the specification of recurring program implementation in a reusable form. Fred does not provide full automation in code generation, but it supports domains where full automation is unachievable or undesirable, such as in the implementation of design patterns and specialization of white-box frameworks.

The way of specifying recurrence in Fred resembles the way DSLs are specified by feature models and pattern languages. Whereas feature models model the problem space of some domain, Fred captures patterns in the solution space. Despite the differing levels of abstraction, the approaches are syntactically similar, and thus it might be reasonable to investigate the possibility of combining the approaches. Given the code generation facilities offered by Fred, a unified approach would allow modeling of both the problem and solution spaces of a given domain as a single DSL, permitting a seamless tool-supported transition from requirements to code.

We aim in a system that would provide high degree of code generation and structural validation, still maintaining its ease of use and applicability to a wide range of technologies such object-oriented framework specialization, instantiation of design patterns and coding idioms, and application of generic architectural styles. The approach in itself is not tied to any particular problem or solution domain, although the implementation is likely support only a restricted set of programming languages.

Chapter 2 provides motivation for this paper by discussing the role of code generation in software engineering and in particular, arguing that full automation is not always enough. Chapter 3 briefly outlines the three existing approaches. A sketch for a unified model is constructed in Chapter 4. Chapter 5 concludes the paper.

2. AUTOMATION IN SOFTWARE CONSTRUCTION

Generative programming aims at full automation in software construction [CzE00]. A generative programming tool assumes a high-level program specification, and is able to generate a software component based on that specification. Typically, the process of writing the input specification is separate from generating the output. Thus, generative programming tools are black boxes that perform input-output transformations. Like compilers, they do not support any interference in between. Preprocessors, template programming and program generators all fall into this category.

The classical generative approach results in total isolation from the produced code, which is excellent, assuming we get full automation. However, sometimes it's not possible – or even desirable – to achieve full automation. This is especially true with software patterns and white-box frameworks; it would be nice to get generative support for design patterns, but patterns cannot be generated in isolation. Instead, they intertwine with the rest of the code. Similarly, white-box frameworks are open by their nature. We can't specialize such a framework by giving a featural description, simply because being white-box it supports unanticipated features. Still, there is a desperate need for code generation for these domains too fuzzy for full automation. Moreover, because of the human component, partial code generation must be complemented with code validation, and is even harder to achieve than full automation.

3. EXISTING TECHNOLOGIES

Fred

Fred (Framework Editor) is a software development environment prototype for Java being developed since 1997 at the Tampere University of Technology and University of Helsinki. Fred helps to capture recurring textures within software in a form that supports systematic generative tool support. Suitable areas of application range from implementation patterns to architectural conventions and white-box framework specialization [Hau02, Vil01]. The tool has been discussed in [Hak01a, Hak01b] and demonstrated in [Hak01c]. It is available for download with a tutorial at http://practise.cs.tut.fi/fred.

In Fred, recurrence is stored in hierarchical structures called *patterns*. Being very implementation-oriented, these should not be confused with design patterns, although implementation-oriented variants of design patterns can be represented in Fred.

A Fred-pattern is essentially a very fine-grained implementation-oriented DSL, presented as a tree with cross-references. During software development, the user instantiates the pattern by walking through the tree under the guidance of the tool, resulting in generated code and documentation. Moreover, instructions on instantiating the pattern are generated on the fly. As a result, Fred releases the user from tedious programming tasks, but at the same time promotes learning by doing.

A minimal Fred-pattern is depicted in Figure 1. The pattern is used in Fred to generate accessor methods for member variables. The pattern consists of *roles* that will be bound to concrete implementation elements during the pattern instantiation. Roles have cardinalities and syntactic constraints in the form of cross-references.

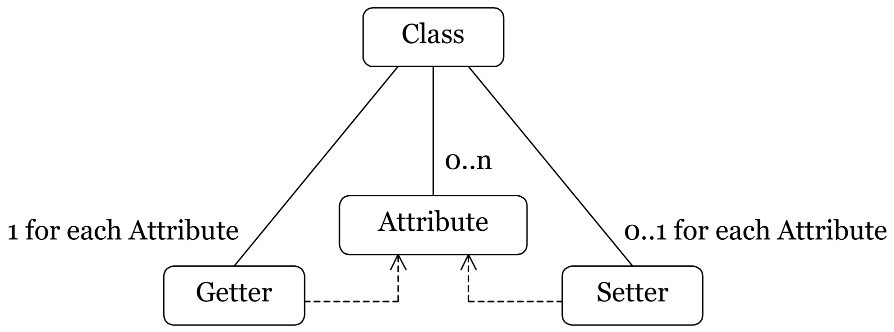

Figure 1. A minimal Fred-pattern

This diagram presents a grammar for a small DSL. Although suppressed from the diagram, semantic information is attached to the nodes to provide code generation, constraint checking and adaptive documentation facilities. E.g. after typing in a string field "name" in a class, Fred is able to provide suggestions like "Provide a setter method for the field *name*" to the developer. Like the instructions, source code can be generated on demand. Following the tasks provided by Fred eventually results in a complete application of the pattern.

Feature Models

A feature model represents and classifies instances of a particular concept providing a structural representation of the possible properties (i.e. features) of the concept instances. Feature models are used in describing variability and commonality of software components within a family of software artefacts. However, although they model variability within software they don't imply any specific means of implementing that variability. Thus, feature models present a convenient way of describing software product or system families at a high level of abstraction.

A feature model is represented as a hierarchically structured diagram. The nodes of this diagram are called *features*, and they present individual configuration points within the family. Features range from high-level requirements to implementation-specific configuration details.

Feature diagrams are used in deriving *featural descriptions* of instances of the family. This process involves traversing through the feature diagram in a well-defined manner binding each traversed feature to some specific configuration. The traversal must proceed according to the syntactic rules of the diagram and each binding must adhere to the semantic description of the particular feature. The result is a set of bindings that uniquely describes a certain piece of software.

Feature Models, Fred and Pattern Languages

Feature diagrams and Fred-patterns are both used in deriving concrete software artefacts from generic descriptions. They differ in their level of abstraction however; where Fred concerns the recurring implementation strategies, feature models attempt to be abstract in this sense.

Despite this, the methods are syntactically very similar. They provide similar means of presenting a hierarchically structured DSL; the tree of features effectively corresponds to the tree of roles. Both approaches also incorporate grammatical constraints – hard dependencies between the elements of the language.

There are syntactic differences too, most importantly the notion of cardinality. This concept essential to Fred is lacking from the feature models in an attempt to maintain them independent of any structural information of the solution space.

All in all, the approaches differ in two aspects; 1) their intended use (level of abstraction), and 2) the expressive power (i.e. the set of languages they can represent).

There is yet another related approach, however. It has turned out that the way patterns are applied in Fred resembles the way pattern languages [Ale79] are applied, although Fred-patterns are of much smaller scope than pattern languages. In essence, Fred-patterns are fine-grained pattern languages, the same way they can be conceived as fine-grained feature models.

The three approaches are far from being identical, but the similarities suggest that the works are characterizing a similar phenomenon. Table 1 outlines these approaches.

Table 1. Terminology and features of the three related approaches

	Feature model	*Fred pattern*	*Pattern language*
Primary elements	features	roles	patterns
Primary structure	tree	tree	graph
Grammatical constraints	requires and mutual-exclusion constraints, default-dependency rules.	cardinality and positive, negative, hard and soft dependencies.	informal annotations

	alternative and or-features	alternative roles*	
Language sentence	featural description	pattern instance	sentence
Example domain	Matrix computation components [CzE00]	Java MVC-framework [Hau02]	Shopping malls [Ale75]

*) Some of the grammatical constraints are not yet implemented in the latest Fred release.

All the approaches are ways of presenting a grammar for a DSL as a graph. Feature models and Fred model some primary structure within the graph as a tree, augmenting it with cross-references (Fred-patterns have also been described using graphs instead of trees, see [Hak01b]). The cross-references are called constraints in feature models and dependencies in Fred.

4. UNIFIED APPROACH

Fred has been promising in supporting the reuse of software patterns, white-box frameworks and generic architectures such as EJB [HaK02]. However, Fred-patterns mostly model recurrence in the solution space (the code), instead of the problem space (the requirements). This results in the inability to support high-level specifications that would disregard the actual implementation.

Feature models suggest the notion of vertical constraints to map high-level specification features onto implementation-level features. During the Fred project the notion of cardinality has proven to be valuable in modeling recurring patterns in the solution space, even though cardinality may not play an important role (or it may be circumvented) in modeling the problem space. Hence, lacking the concept of cardinality, feature models themselves are inadequate in providing support for code generation.

Our vision is to use Fred machinery to support specifications of various levels of abstraction. Vertical constraints binding the featural model of the problem space to the patterns of the solution space, could serve as a foundation for the attempt.

We are currently carrying out a project that continues the development of Fred by generalizing its implementation to support patterns, feature models and pattern languages of arbitrary domains and granularity. The terminology for the new model is adopted mostly from the field of pattern languages, reusing existing Fred-terminology whenever possible. Thus, a DSL will be called a pattern language, being composed of patterns.

The main problem in combining the approaches rises from the fact that creating a language capturing a given domain all the way from high-level requirements to variable implementation details would be an overwhelming effort. The problem could be sized down considerably however, by making it possible to reuse existing patterns and pattern languages in creating new pattern languages. The reuse mechanisms planned for Applause are specialization and dynamic composition of pattern languages.

Pattern Language Specialization

A pattern language is a description of a certain domain, such as a system or product family. Typically, it is applied to get a description of a specific instance of that family. However, we consider this as a special case. In general, application of a pattern language corresponds to refining the original language to a narrower domain, and results in a new pattern language. This language may indeed cover only a single instance of the family, but that is not necessary. Thus, applying a pattern language is called *specialization*. This is the fundamental reuse mechanism for pattern languages. E.g. a pattern language describing the structure of a design pattern may be specialized to get a pattern language for a specific implementation pattern.

Foundations of Approach

Pattern language is a structure of patterns. The patterns are organized in a parent-child hierarchy with cross-references. Figure 2 depicts a pattern language of four patterns, with pattern Server as the root, and a cross-reference from Handler to Request.

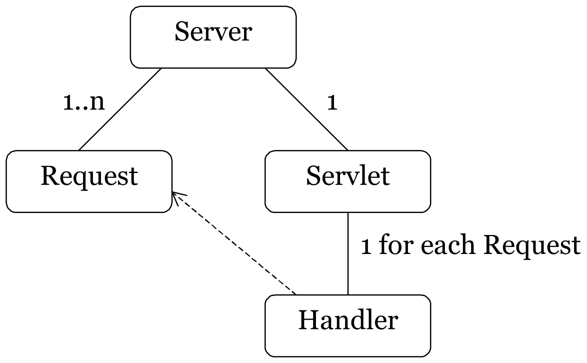

Figure 2. A pattern language for creating Java servlets

This pattern language is an abbreviated description on how to create server applications that handle multiple types of requests from clients. Although suppressed from the diagram, each pattern encapsulates a description of certain part of this domain. E.g., the pattern Request stands for the requests the server should be able to handle. Servlet-pattern describes how to implement the server using Java Servlet technology. It is supplemented by Handler, which describes the handler methods that should be provided for each request.

In general, the patterns provide a vocabulary for the domain, and the relationships between patterns form a grammar for composing sentences describing sub-domains or specific instances of the original domain. In other words, pattern language is always applied to create a new language. Creating a new pattern language corresponds to refining patterns of the original language according to the semantics of the original patterns and the grammatical rules of the original language.

Figure 3 presents another pattern language. It describes a servlet that is able to respond to two types of requests (log in and fetch). Clearly, this language describes a sub-domain of the previous pattern. It has been created according to the grammatical rules of the previous language. Patterns LogIn and Fetch refine the pattern Request of the previous pattern language. Similarly, the original Handler-pattern is refined for the two types of requests.

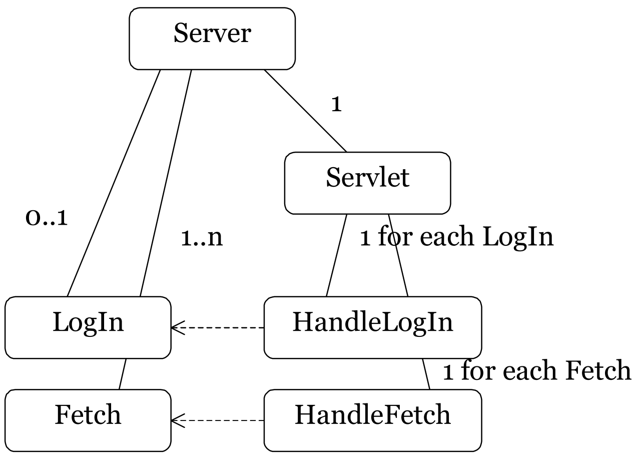

Figure 3. A refined pattern language for Java servlets

This language can be further refined, however. The logging functionality is marked as optional, and there can be several kinds of fetch-requests. A new language may introduce a fixed set of requests, or even dictate the exact implementation of the handler methods. In case the domain is completely fixed and thus cannot be further refined, the pattern language effectively describes a single application. In other cases, the language encapsulates architectural configuration knowledge of a certain domain.

If we look at three patterns on the left side of the previous figure, we see something that looks like a feature model with cardinalities. A more specialized language with no cardinalities would correspond exactly to a conventional feature model. Application of a pattern would then correspond to selecting the particular feature. E.g., the logging functionality could be selected by applying the LogIn pattern.

The other patterns in the figure describe low-level implementation features. As we have dependencies between the sides, the high-level decisions are effectively mapped to low-level implementation strategies. With similar bindings, it is possible to bridge problem space to solution space, and eventually produce code based on the low-level pattern descriptions.

The approach does not impose any particular requirements on the patterns. Our focus has been in working on the mechanical interpretation of pattern languages and thus we're trying to leave the semantics of patterns as open as possible, allowing arbitrary extensions to the approach. However, three important kinds of patterns can be identified based on what kind of information they contain and how they should be applied. These are features, roles and constraints. They are all considered patterns in our approach; they all describe a domain, but with different kind of content.

Features

Features correspond closely to the features of feature models. They encapsulate some configuration knowledge. Refining a feature means narrowing the variability captured by the feature. Traditionally a feature in a feature model describes a point and bounds of variation, and a featural description (the result of the application of a feature model) would select a unique value within the range of variation. In general, the feature in a refined language must define a sub-domain of the original feature, but it does not need to be a single value.

In the previous example, the patterns LogIn and Fetch are examples of features, refining the Request feature.

Roles

The second important category of patterns are roles. They are descriptions of program elements such as classes, methods, fields and so on. A pattern language may leave the actual program element open, in which case the pattern may be further refined in subsequent pattern languages, or bind the role to a specific piece of code.

The description of a role should encapsulate the knowledge on writing the required piece of code, or generating the code automatically. The description can include templates and scripts that make use of the dependencies within the pattern language, allowing code and instructions to be generated according to the other generated elements and selected featural configuration. E.g., the code template for the Getter method role in Figure 1 could be “return #Attribute#;”, expanding to something like “return name;” during the development time.

In our previous example, the pattern Servlet is a class role, and patterns HandleLogIn and HandleFetch are method roles.

Constraints

The third class of patterns is constraints. They describe conditions on the program elements bound to roles.

There are no constraint patterns in the previous example, but it could be augmented with constraints. E.g., it would be natural to add a constraint demanding the servlet class (bound to role Servlet) to extend the HttpServlet base class declared by Java Servlet API. This requirement could be imposed as a constraint pattern, attached as a child of the Servlet role.

To achieve generative tool-support, there must be a tool-supported format for defining the semantics of patterns. The tool could then use a pattern language as the basis of guiding the developer through developing a new application for the domain, or defining a more restricted sub-domain. The developer is responsible of applying the pattern language, but the tool makes sure that the pattern language application is syntactically correct, and enforces the semantic constraints embedded in pattern descriptions. Scripts provide automatic and semi-automatic code generation, and architecture-specific violations could be reported at compile-time based on constraint descriptions.

At the heart of this are the features that are used in gathering the featural description of the system under construction. Depending on the nature of the pattern language the system could be generated automatically based on the choice of features, or the features could be used as the basis of co-operative software development between the tool and the developer.

Pattern Language Development

The approach makes no distinction between deriving software systems based on pattern languages and deriving new pattern languages from old ones. An important and immediate consequence is that pattern languages itself can be developed exactly the same way as the applications are derived from the pattern languages. Thus, if we have a generative tool for software, we also have a generative tool for describing feature models, design pattern variants, architectures, and everything that can be captured by the kinds of pattern languages proposed.

The discussion has also discovered why cardinalities do play an important role in after all. Even though cardinalities do not play an important role in conventional feature models, a generic DSL would be hard to write without them. E.g., without cardinalities, the language in Figure 3 could not be considered a specialization of language in Figure 2, and thus no tool support could be provided in that language.

A continuation to our previous example is depicted in the Figure 4. Patterns Server, LogIn, Search and Download essentially define a simple feature model with no cardinalities. The roles Servlet, HandleLogIn, HandleSearch and HandleDownload can be used to generate code for the application based on the selected choice of features. The pattern language is a refinement of a more general language, the one we saw before, and Search and Download are actually refinements of Fetch-feature. Thus, a feature model can exist without cardinalities, but they are central to general-purpose pattern languages.

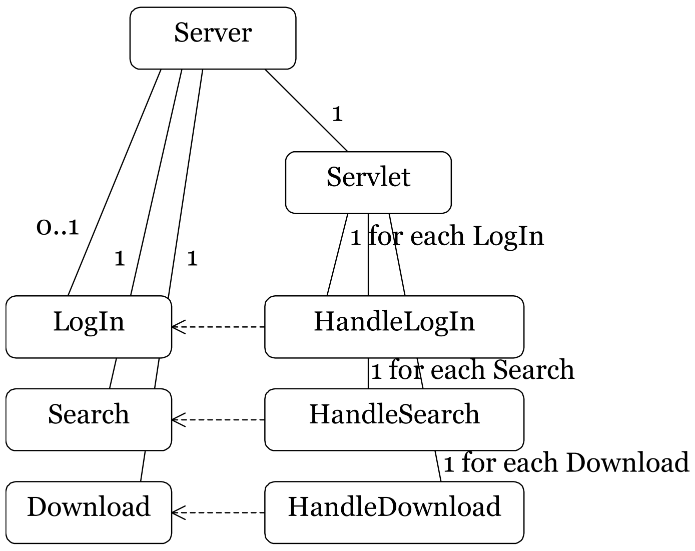

Figure 4. A pattern language with limited cardinalities only

It is possible for a pattern language to combine multiple existing pattern languages. E.g., continuing with our example, it would have been possible to create the original example as a refinement of two languages – a pattern language for servers in general, and a pattern language for servlets. The figure below depicts these primordial languages, however omitting many details as in previous examples.

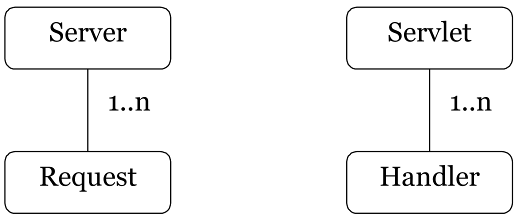

Figure 5. Simple pattern languages that could have been used as the basis of composing the language in Figure 2

Dynamic Composition of Pattern Languages

Pattern language specialization alone is not sufficient mechanism for reusing pattern languages, however. Consider we were supposed to describe a pattern language for web servers, as in previous examples, but wanted not to restrict the language on any particular implementation technology, like Java servlets. In general, sometimes we would like to leave a part of the pattern language open, to be refined later, in the subsequent specialization of the language. For this purpose, another mechanism is required.

Dynamic composition of pattern languages means that in certain pattern language we leave some parts unspecified by introducing a placeholder so that another pattern language can augment the original specification at later time. This requires that we can somehow specify the boundary between two pattern languages, and the required characteristics of these parts. Thus, a typing system for pattern languages is required.

Figure 6 repeats our original example, but this time using dynamic composition. The pattern marked with dashed border indicates a slot to be fulfilled by some other pattern language. A dashed arc in the diagram indicates that this pattern language is able refer to the Request pattern of the host language by the name R.

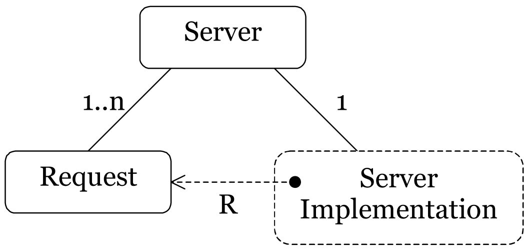

Figure 6. A pattern language that makes use of the dynamic composition

Another pattern language is presented in Figure 7 that may be used in place of the server implementation of the previous pattern language. This language describes a server implementation using Java servlets.

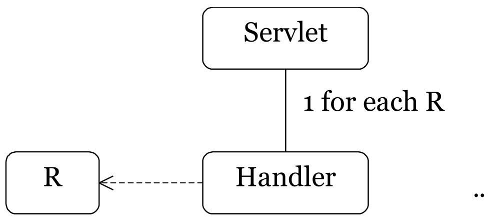

Figure 7. A pattern language that can be used to fulfill the slot of the language in Figure 6

The pattern language defines an external pattern R. This is used in gluing the two pattern languages together. When using this pattern language to fulfill the slot in the previous language, Request-pattern stands for R. As a result, we get a language that looks like our original example, but has resulted from the dynamic composition of pattern languages. This allows us to specify multiple variations of server implementation independently of the original pattern language. We can even apply more fine-grained decomposition by specifying the contents of the Handler-pattern as a separate pattern language. This would allow different kinds of handler implementations according to differing needs. E.g., handlers could be implemented as ordinary methods, or utilizing the Command design pattern [Gam94] (which in turn could be presented as another pattern language).

5. CONCLUSION

Feature models and pattern languages have much in common. This paper has outlined our attempt in capturing the essence of these approaches to provide systematic architectural tool-support. The approach is based on the experiences of Fred-project, extending the DSL-processing machinery already present in the Fred-tool. The resulted model itself is independent of any particular problem or solution domain in the same sense feature models or design patterns can be applied in variety of domains.

The promise of the unified approach is in bridging languages of problem space to the languages of solution space, providing seamless, tool-supported and traceable transition from requirements to code. However, the problem in combining the approaches lies in the complexity and size of the resulted DSLs, and must be confronted by DSL-level reuse mechanisms.

We have briefly outlined ideas on pattern language specialization and dynamic composition, as mechanisms for specifying domain specific languages by reusing others. It is our aim to provide a framework that would allow development of reusable pattern language libraries capturing design abstractions in the same way object-oriented frameworks are used in capturing architectures at implementation level.

The approach itself is not tied to any particular programming language. At the time of writing, we have a tool prototype supporting pattern language specialization, and are continuing to investigate dynamic composition. Fred 2.0, incorporating these features, as well as a Java-framework for extending support for different programming languages and solution domains, is to be released in the first quarter of 2003. The release incorporates support for Java and XML.

REFERENCES

[Ale75] Alexander, C., Silversteim M., Angel S., Ishikawa S., Abrams D., The Oregon Experiment, Oxford University Press, 1975.

[Ale79] Alexander C., The Timeless Way of Building, Oxford University Press, New York, 1979.

[CzE00] Czarnecki K., Eisenecker U. W., Generative Programming: Methods, Tools and Applications, Addison-Wesley, 2000.

[Gam94] Gamma E., Helm. R, Johnson R., Vlissides J., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

[Hak01a] Hakala M., Hautamäki J., Koskimies K., Paakki J., Viljamaa A., Viljamaa J., Annotating Reusable Software Architectures with Specialization Patterns. In: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01), Amsterdam, August 2001, 171-180.

[Hak01b] Hakala M., Hautamäki J., Koskimies K., Paakki J., Viljamaa A., Viljamaa J., Generating Application Development Environments for Java Frameworks. In: Proceedings of the 3rd International Conference of

Generative and Component-Based Software Engineering (GCSE'01), Erfurt, Germany, September 2001, Springer, LNCS2186, 163-176.

[Hak01c] Hakala M., Hautamäki J., Koskimies K., Paakki J., Viljamaa A., Viljamaa J., Architecture-Oriented Programming Using Fred. In: Proceedings of International Conference of Software Engineering (ICSE'01), Toronto, May 2001, 823-824. (Formal Research Demo)

[HaK02] Hammouda I., Koskimies K., Generating a Pattern-Based Application Development Environment for Enterprise JavaBeans. Accepted for presentation in Computer Software and Applications Conference (COMPSAC'02) Oxford, August 2002.

[Hau02] Hautamäki J., Task-Driven Framework Specialization – Goal-Oriented Approach. Licentiate thesis, Department of Computer and Information Sciences, University of Tampere, January 2002.

[Kan90] Kang K., Cohen S., Hess J., Nowak W., Peterson S., Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, November 1990.

[Vil01] Viljamaa A., Patter-Based Framework Annotation and Adaptation – A Systematic Approach. Licentiate thesis, Department of Computer Science, University of Helsinki, June 2001.