Architecture Reconstruction in Practice

Claudio Riva
Nokia Research Center
P.O. Box 407, FIN-00045, NOKIA GROUP, Finland
Tel: +358 50 4837403, Fax: +358 7180 36308
claudio.riva@nokia.com

Abstract: The description of the software architecture should communicate the essential decisions that have been taken during the design of the software system. Architecture reconstruction is a reverse engineering activity that aims at recovering past decisions that are either unknown (because not documented or the original developers have left) or new (because originates from the system's evolution). The reconstruction is performed by examining the available artefacts (documentation, source code, experts) and by inferring new architectural information that is not immediately evident.

In this article we describe our architecture reconstruction method and the environment supporting it. The method emphasises the importance of the architecturally significant concepts (essential for the architects) and the domain specific knowledge. They are considered first class entities in the reconstruction process from the initial stages. We believe that focusing the reconstruction in this way we can produce quality information for the architects.

Key words: Reverse engineering, architecture reconstruction, software architecture

1. INTRODUCTION

Similar products are usually organised in *product families* [21], a set of products that have common requirements and, consequently, share common features, chuck of functionality, architectural concepts or code typically in the form of components. The product family strategy aims at reducing the

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35607-5_15

J. Bosch et al. (eds.), *Software Architecture*

development, maintenance, support and marketing costs by reusing some of the system parts across several products. In [19], we have proposed a method to describe the architecture of a large product family and to manage its evolution. The method is based on two separate concepts: the *reference architecture* and the *configuration architecture*. The reference architecture describes the architectural style that is valid for all the products of the family. The style includes the requirements, rules and patterns that are significant at the architectural level. The architects can derive the software architecture for the single products from the reference family architecture. The configuration architecture describes the organisation of the product family features. Features can be common for the whole family or specific for the single products. The configuration architecture specify the rules how to map the product family features into the various products, thus, allowing to model commonality and variability.

According to this method, the evolution of the product family is driven by the evolution of the reference and configuration architectures. The reference architecture slowly evolves by capturing and incorporating new architectural significant requirements of the new products. New products are added to the family by extending the configuration architecture with new features and new composition rules.

In practise, this approach is carried out by a combination of forward and reverse engineering. Forward engineering activities are necessary to develop the new features of the products starting from their requirements. Reverse engineering activities recover the concrete implementation of the products, monitor the organisation and implementation of the features and verify the conformance to the architectural rules. Architecture reconstruction plays a key role in the product family evolution. A clear comprehension of the product architectures allows us to continuously evolve the family by aggregating the common features in the family architecture and monitoring the implementation of the products.

This article describes the reverse engineering [3] method that we use to comprehend the actual implementation of the products. Our focus is mainly on the architectural significant aspects of the products. In the literature, this flavour of reverse engineering has been called *reverse architecting* [16] or *architecture reconstruction* [14].

The main driver of our work is to enable the software architects to analyse the structural dependencies that exist in large software systems, as presented in our previous work [26]. Those dependencies are often unclear, hidden in the details of the implementation or just not shown at the right level of abstraction. Hence, the typical quest for *"the big picture"* of the system that leads to a clear understanding of the interactions among the major components. One aspect that is often difficult to grasp with a bottom-

up approach (like reverse engineering) is the set of design decisions that have been made to implement the features of the system. Our approach aims at recovering them.

2. RELATED WORK

In our previous work, we have related our work with other research in the field of architecture reconstruction [27] and dynamic analysis [28]. In this section, we report the major shortcomings that we have detected.

Kazman et al. propose an iterative reconstruction process [8] where the historical design decisions are unveiled by empirically formulating/validating architectural hypothesis. The approach is supported by the Dali workbench [15][14]. Dali allows the user to create a source code model in a SQL database. The user can then base the abstraction process (mainly a grouping activity) on a set of queries executed in the database. In our experience, the select/group paradigm is not expressive enough to model the architectural abstractions. In our approach, we have chosen Prolog for the abstraction phase in order to have a more expressive mechanism than SQL. They also point out the importance of modelling not only system information but also a description of the underlying semantics [8]. In our method, the first phase aims at clarifying the semantics of the concepts involved in the reconstruction.

Krikhaar et al. [16] adopt the paradigm extract/abstract/present for architecture reconstruction and base all the reconstruction operations on the Partition Relation Algebra [7]. In our approach, we generalise the method to any architectural style by introducing an additional activity that takes care of focusing the reconstruction on the most important architectural aspects for the architects.

Finnigan et al. [6] propose the Software Bookshelf that is a collection of tools for generating software architectures from program sources and presenting them in a Java-based web user interface. The goal is to keep the architectural documentation up to date. The tool has been used to extract the software architecture of Linux operating system [2]. One key feature is the web interface that allows the architects to publish the architectural diagrams on the intranet. In our environment, we have included the web application Venice that allows us to publish the diagrams on the web in UML format.

Murphy et al. [20] propose a reconstruction technique based on the reflexion models. The user starts with a structural high-level view model that is iteratively refined to rapidly gain knowledge about the source code. The technique is based on the definition of a set of mappings between the source code and the high-level concepts. Our technique generalises this idea

enabling the user to define any kind of mappings or transformation of the source code model.

Most of the approaches adopt the extract/abstract/present paradigm and rely on different formalism for conducting the abstraction operations (SQL, Partition Relation Algebra [7], Tarski algebra [9] or simple maps [20]). In our approach, we exploit Prolog as a mechanism for conducting a series of abstraction operations.

The dynamic analysis aims at describing the run time behaviour of a software system. Its contribution should be considered during an architecture reconstruction process. Some attempts have been done to merge the dynamic and static information in a single view: Systä [29], IsVIS [12], Dali [13] and Richner et al. [23].

3. ARCHITECTURE RECONSTRUCTION

The description of the software architecture should communicate the essential decisions that have been taken in the design of the software system. The essential decisions of a design are the ones that are expensive to change and, therefore, the most critical for the development and maintenance of a system. A. Ran [21] defines four categories of design decisions: *concepts* (the way we think of a system, its architectural style), *architecturally significant requirements* (the major concerns that have to be addressed by a proper software architecture), *structure* (the components and their relationships at the right level of abstraction) and *texture* (design decisions at the implementation level that are architecturally relevant, such as design patterns and policies). A software architecture is defined as *"a set of concepts and design decisions about structure and texture of software that must be made prior to concurrent engineering to enable effective satisfaction of architecturally significant, explicit functional and quality requirements, and implicit requirements of the problem and the solution domains"* [21]. Multiple views (such as the "4+1 model" proposed by Kruchten [17] and the architectural views proposed by Hofemeister [11]) are a practical way to effectively communicate the different aspects of the software architecture.

Architecture reconstruction (or reverse architecting) concerns with the task of recovering the past design decisions that the developers made during the development of the system. It is a reverse engineering activity that has to infer the architectural rationale from the available artefacts created by the original developers (who might have left or not documented the architecture). The natural evolution of a software system also introduces new aspects that a reconstruction process can unveil (in this case we can talk of a information discovery process).

Architecture reconstruction is not only a pure data gathering process but requires also a certain amount of reasoning for the selection and analysis of the extracted artefacts. We stress the point that only the correct choice of the architectural concepts (selected according to the system) can deliver a meaningful high level model to the architects. The architectural concepts are first class entities in the reconstruction process from the very early stages.

The output of the reconstruction has to present the different aspects of the model with multiple architectural views. In practice, we aim at delivering the following architectural views:

- Conceptual view: describing the key architectural concepts that are instantiated in the other views.
- Component view: describing the major components, their interfaces and their logical relationships.
- Development view: describing the organisation of the source code files and their relationships (for example, include dependencies).
- Task view: describing the task allocation of the architectural entities and showing the inter task communications.
- Feature view: describing the run-time implementation of a feature at a high level of abstraction.

The views are based on static aspects (captured without running the system) and dynamic aspects (concerning with the run-time behaviour). They are both necessary for the architectural description and they have to be adequately reverse engineering from the implementation.

We can summarise our iterative and incremental process in four steps:

1. Recovery of architectural concepts

The goal of this phase is to recover and clarify the architecturally significant concepts that build the system: the building blocks of the system and the communication infrastructure that enables the components to interact at runtime. These concepts represent the way developers think of a system and they become the terminology of the reconstruction process. The architectural concepts vary from one system to another: in a distributed software system the architectural concepts may be applications, servers, software busses while in an operating system they may be tasks, processes, queues, shared memories, etc. Textures should also be considered at this stage because they hide interaction patterns that are architecturally significant for the reconstruction (like the design patterns).

The outcome of this phase is the conceptual view that describes all the important types of architectural concepts and their relations, and the description of the mappings between the high level concepts and the implementation.

The main source of information is the documentation of the system or informal discussions with the experts. We often find useful to ask the

developers to describe the implementation of the key features of the system. During this explanation, the architectural concepts become evident.

2. Model capture

We build a model of the system whose entities are instances of the concepts identified in the previous phase. A correct choice of the concepts ensures that the model is built at the right level of abstraction. Being mainly a data-gathering phase (instead of a reasoning phase), this task can be easily automated with tools for analysing the system artefacts. Source code is usually the most dependable source of information for the static analysis. We rely on ad hoc analysers (for example written in Perl) based on pattern matching or on commercial programming environments with APIs to the symbol tables (such as SourceNavigator [22]). The documentation, the software diagrams (for example, stored in CASE tools) and the experts can contribute to the creation of the model. For the dynamic analysis, we instrument the system and trace relevant information by simulating particular use cases (for example, using the ThirdEye environment [18]).

3. Abstraction

The model of the previous phase is at a very low level of abstraction. The goal of this phase is to enrich the model with domain specific knowledge that will lead to a high level view of the system (for example, to create the structural description). Known abstractions can be easily added to the model. Unknown abstractions have to be identified by the architects, categorised, named and then stored in the model. The reasoning is carried out manually by the architects and produces to a set of abstraction rules that enrich the model. We point out that the abstraction process is not just an activity of grouping but it is a reasoning process where we infer more abstracted relationships. We specify the abstraction rules with a logical language like Prolog.

4. Presentation

An effective visualisation is essential to communicate the architectural information to the development teams. The architects need to select a particular architectural view and a particular visualisation format: hierarchical graphs, web documents (with hyperlinks), UML [1] logical diagrams and message sequence charts. We use Rigi to visualise hierarchical oriented graphs. It enables the architects to navigate the model, analyse the dependencies and identify new possible groupings to add into the model. In our previous work [25], we have exploited Rigi for this kind of tasks. UML diagrams are a familiar way to convey architectural information to the designers. The tool Venice [31] gives us the support for visualising logical views using a subset of the UML notation (components, packages,

interfaces, inheritance and dependencies) that we have proposed in [27]. Figure 3 shows an example of visualisation in Venice.

We have integrated a message sequence chart visualiser with Rigi to combine the static and dynamic analysis [28].

This process has to be reiterated several times to produce a quality model for the architects. The initial abstraction rules are based on the conceptual view that the developers have of the system and may be different from the real one. New architectural concepts become significant while the reconstruction is progressing and have to be introduced in the model. The data-gathering phase can also be refined by increasing the quality of the extracted information with more powerful analysers (often the extraction is a trade-off between the speed/size and the quality of the analysis).

4. THE LEVELS OF ABSTRACTION

The artefacts of a software system (such as code, design documents, user interface specifications, feature lists) have different levels of abstraction. Reverse engineering is a process spanning from the low levels to the higher ones. We can identify six levels of abstractions (grouped in two categories) that define a scope for the artefacts of the reconstruction process *requirements*, *domain model*, *features*, *architecture*, *design* and *source code* as shown in Figure 1.

We distinguish between the *problem domain* (focused on the user's perspective) and the *solution domain*. The problem domain specifies *what* the system is supposed to do. The solution domain specifies *how* the system achieves what is promised.

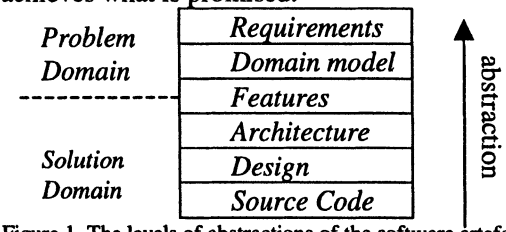

Figure 1. The levels of abstractions of the software artefacts.

The functional requirements are mapped to *features* that the system has to support. A feature is a "coherent and identifiable bundle of system functionality" [30]. The features are the highest elements of abstraction we can decompose a system in the solution domain. The elements of the lower levels are responsible for implementing those features. In particular, at the *architectural* level we are interested in modelling the structure of the

architecture description (see Section 3), in order to show what components are involved and how they interact. At the *design* level, we model the internal implementation of the architectural elements. Each component is clearly specified at the design level with a particular formalism (such as the object-oriented paradigm). The lowest level of abstraction is the *source code* level. The abstract syntax trees (AST) model the information at this level for reverse engineering purposes.

The features represent the contact point between the problem and the solution domain [30]. At this level, marketing people and developers can speak a common language and they can understand each other. Features are used to advertise the system and have to be implemented by the developers. Although a feature might not have a one to one mapping with the architectural concepts, we believe that a complete reverse engineering process should aim at the identification of the system features and their interactions. Turner et al. call this process *"feature oriented reverse engineering"* [30].

5. AN EXAMPLE

We demonstrate the reconstruction method with an example that is the simplification of a real case. The real case is taken from a family of products for telecommunications where time to market usually forces the developers to quickly instantiate new products from the family disregarding their documentation. The proposed architecture reconstruction method can help the developers analyse the architecture of the products [25].

5.1 Architectural concepts.

The system is component based. Components (implemented by a set of C functions) represent computational units or resource controllers and offer well-defined services through their interfaces. The communication among the components is achieved with the exchange of asynchronous messages on a software bus. There are two OS primitives for registering on the bus and sending messages:

- register(ID) – primitive to register a component "ID" on the bus.
- send(d, m) – primitive for sending the message "m" to the component "d".

When the components are initialised, they register themselves on the bus with a unique identifier that is statically assigned at compile time. The identifier is used by the "send" and "receive" primitives.

One key issue of the architects is to manage the organisation of the components so that they can collaborate to implement the system features.

Each message exchange between two components creates a dependency that has to be taken into account by the architect. In a system with hundreds of components the dependency graph becomes rather complicated. For this task, the architects need (1) the component view that shows the logical organisation of the components in packages and their dependences, (2) the execution view that shows how the components interact and (3) the development view that shows how implementation of the components.

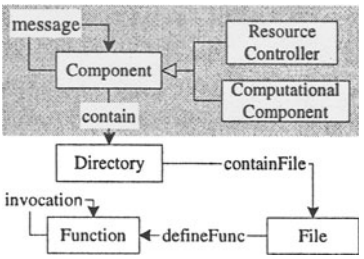

Figure 2. The architectural concepts and their relationships.

The diagram in Figure 2 shows the major concepts that we think are architecturally relevant to the architects. We distinguish between the concepts that are visible in the implementation (like *Directory, File, Function* and their relationships) and concepts that can be inferred from the previous ones (like *Component, message* and *contain*). The latter ones are located in the grey area of the diagram.

5.2 Extraction of Static Information

We extract a source code model with an ad hoc analyser and present the output as a set of Prolog facts.

Below is a sample of the information extracted by the code analyser.

```
containDir('/gui','/gui/VoiceCall').
containFile('/gui/VoiceCall','/gui/VoiceCall/mainApp.c').
defineFunc('/gui/VoiceCall/mainApp.c','init').
defineFunc('/gui/VoiceCall/mainApp.c','makeCall').
invocation('init','register',['VOICE_CALL']).
invocation('makeCall','send',['CALL_CTRL', 'SETUP']).
invocation('makeCall','send',['CALL_CTRL', 'CALL']).
invocation('makeCall','send',['NET_CTRL', 'ALERT']).
```

For instance, the first line define a containment relationship (*containDir*) between the directories '/gui' and '/gui/VoiceCall'. The facts about the function calls (*invocation* relationship) contain also the details about the parameters of the call.

5.3 Abstraction

The extracted source code model is at a very low level of abstraction and represents we use it to infer more abstracted information about high-level concepts (the ones in the grey area of Figure 2). The abstraction process can be divided in three steps: *model refinement, injection of composition rules* and *view selection.*

Model Refinement

The first step is to refine the model by inferring the new relationships that are not present yet (the ones in the grey area of Figure 2). The Prolog language allows us to formally specify the new relationships. Below there are two Prolog preposition that define the *message* and *register* relationship.

```
(1) message(Src, Dest) :- invocation(Src, 'send', List), nth0(0, List, Dest).
(2) register(Dir, ID) :- containFile(Dir,File), defineFunc(File, Func),
        invocation(Func, 'register', List), nth0(0, List, ID).
```

The proposition (1) defines a *message* relationship between the *Function* that sends a message and the component's identifier of the message by selecting all the "send" function calls.

The proposition (2) defines a *register* relationship between the *Directory* that registers a component to the bus and the component's identifier. This relationship is auxiliary for the following abstractions.

Injection of Composition Rules

This step concerns with adding the part-of relationships to the model to create a hierarchical structure in the model. The composition rules specify how the source code elements are grouped to form subsystems or more abstracted entities. The clustering activity is usually driven by the documentation for known groupings or by the system experts for unknown ones. Below there is an example in Prolog where we define four new components.

```
contain('VoiceCall','/gui/VoiceCall').
contain('DataCall','/gui/DataCall').
contain('CallController','/ctrl/CallCtrl').
contain('Network','/ctrl/NwtCtrl').
```

The previous components are then grouped in component sets according to their functionality.

```
contain('CallServices','VoiceCall').
contain('CallServices','DataCall').
contain('GUI','CallServices').
contain('Resources','CallController').
contain('Resources','Network').
```

View selection

The architects need to select a particular architectural view over the model that we have created so far. To define a view, we have (1) to select its representation format and (2) to define the set of relationships that have to be projected in it. We can represent the development view with a typed directed graph. We define a new relationship *edge* with three parameters: the source node, the destination node and the type of the edge. Then, we select the *contain* and *containFile* relationships from the model. Below there is the Prolog code.

```
edge(X, Y, 'contain') :- contain(X, Y).
edge(X, Y, 'containFile') :- containFile(X, Y).
```

To create the component view we need to compute the high level dependencies among the components. We represent the logical view with typed oriented hierarchical graphs. This is achieved by (1) defining a *grouping* relationships that describes the hierarchy, (2) define the set of relationships of the graph, (3) compute the transitive closure and (4) create the graph.

Below there is the Prolog code that defines the *grouping* relationship and the *relation* relationship.

```
grouping(X,Y) :- contain(X,Y).
grouping(X,Y) :- containFile(X,File), defineFunc(File,Y).
grouping(X,Y) :- register(X,Y).
relation(X,Y) :- message(X,Y).
```

We can calculate the transitive closure with an auxiliary function *trans* define by the following Prolog code:

```
trans(Rel, X, Y) :- P=..[Rel,X,Y], call(P).
trans(Rel, X, Y) :- P=..[Rel,X,Link], call(P), trans(Rel,Link,Y).
```

We can then create the graph by defining a *hierarchy* relationship that is basically the *grouping* relationships. The edges of the graph are obtained by the union of the edges of the *relation* relationship and the ones obtained by the transitive closure. Below there is the Prolog code.

```
hierachy(X,Y) :- grouping(X,Y).
edge(X,Y) :- tran(grouping,X,T1), tran(grouping,Y,T2), relation(T1, T2).
edge(X,Y) :- relation(X,Y).
```

5.4 Visualisation.

Figure 3 shows the component view using Venice. The packages have been created according to the *hierarchy* relationship. The edges show the high level dependencies that exist among the packages. The user can select at which level of detail for the visualisation of the edges.

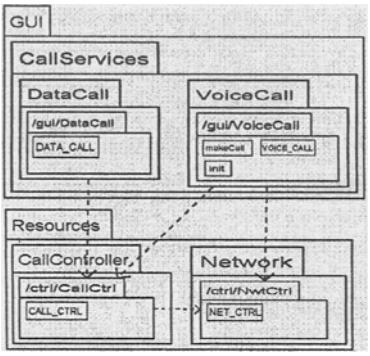

Figure 3. The component view (Venice).

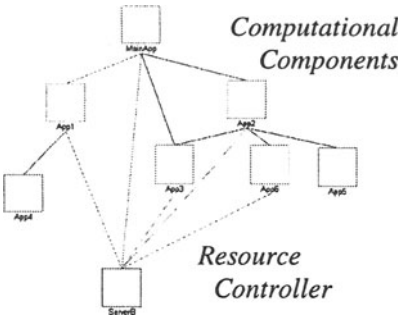

Figure 4. Dependency analysis with Rigi.

Figure 4 shows the visualisation of a typed oriented graph using Rigi. The graph shows the dependency between a set of components and a particular resource controller. This view has been generated by navigating the model using Rigi as we have presented in our previous work [27].

5.5 Extract dynamic information.

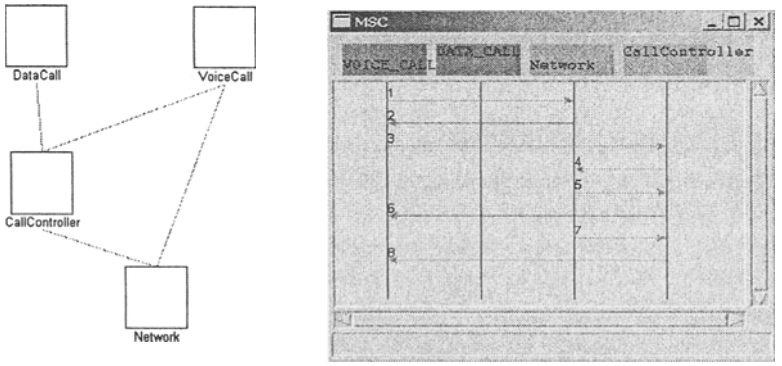

Figure 5. The combined static and dynamic visualisation for the feature view.

Dynamic analysis is necessary for analysing the behaviour of the system and for creating the feature view. The extraction of dynamic information is conducted by (1) instrumenting the source code, (2) executing a set of scenarios and (3) collecting the traces. The architectural concepts of phase 1 drive us in the choice of the correct instrumentation. We choose to trace the calls to the "send" and "register" primitives that are architecturally significant for our analysis. We have used the ThirdEye environment for instrumenting the code. We usually select them according to the system's features that we want to analyses. The traces are then converted to Prolog facts and then visualised with a Message Sequence Chart visualiser integrated with Rigi as presented in [28].

6. EXPERIENCES WITH THE APPROACH

We have applied the architecture reconstruction method on several embedded software systems developed by Nokia. The systems are developed in C or C++ and contain about hundreds thousands lines of code. We report here the major impressions that we have collected:

The first phase of the approach allows us to set the focus the architecture reconstruction activity. The selection of architecturally significant concepts ensures that we will produce useful information for the architects with the right level of abstraction. In this way, we can recover the essential design decisions of the system. Being the process iterative and incremental, we can start with a simple set of concepts and then enrich it when we increase our understanding of the system.

The analysis of the source code does not seem to be a big issue anymore. Nowadays, we can find powerful analysers that extract quality information with simple APIs for accessing their symbol tables. The only hassle is to program the tools to extract the information we need. In the case these analysers do not deliver the information we need, Perl (or just the *grep* utility) are still the best choice.

The abstraction phase allows us to inject domain knowledge about the system and to increase the level of abstraction of the model. This information usually comes from existing design documents and experts (architecture recovery) or we have to create it from scratch (architecture reconstruction). Prolog gives us the capability of formally specifying the abstraction rules and to reuse them, with little changes, for different products of the family. Prolog also offers the possibility of calculating architectural metrics and identify patterns in the model (two aspects that have to be exploited in the future work).

Visualisation plays an important role for understanding the architectural model during the whole process. We visualise the model as a hierarchical typed graph with Rigi and Venice. This allows us to intuitively navigate the model and manipulate the architectural information. Architects find very useful to navigate the software models using the graph paradigm. This feature is often missing from the traditional CASE tools.

The reverse engineered architectural models show the actual implementation of the system. The architects found this information very valuable and they appreciate that fact that is presented using multiple architectural views. These views can be used during the architectural reviews and during the software architecture assessments of a system.

7. CONCLUSIONS

The presented approach for architecture reconstruction allows us to extract valuable information for the architects who are mainly interested in the high level architecturally significant concepts of a software system. The method stresses the point of selecting the correct concepts that will drive the reconstruction process. Abstraction plays a key role in the whole process and it is addressed in all the phases of the method. The supporting environment also gives us the correct set of tools to achieve the task.

Multiple views, like for the phase of architecture modelling, are the means to convey the architectural information among the development teams. In this article, we have just mentioned the feature view that we have not extensively exploited yet. The feature view enables us to describe the implementation of the system features in a very compact and expressive form. Our future work will concentrate on elaborating this concept.

ACKNOWLEDGEMENTS

This work is supported by the CAFÉ Project (from Concept to Application in system-Family Engineering), EUREKA 2023/ITEA - ip00004, http://www.extra.research.philips.com/euprojects/cafe/.

REFERENCES

1. Booch G., Rumbaugh J. and Jacobson I., *The Unified Modelling Language User Guide*, Addison-Wesley, 1999.
2. Bowman T., Holt R. C., and Brewster N. V., Linux as a Case Study: Its Extracted Software Architecture, *Proc. of the International Conference on Software Engineering (ICSE '99)*, Los Angeles, May 16-22,1999.
3. Chikofsky E. J. and Cross J. H. II., Reverse Engineering and Design Recover: A Taxanomy. *IEEE Software*, Jan. 1990.
4. Dagstuhl Middle Model (DMM), proposal, http://www.site.uottawa.ca/~smarchen/dmm
5. Demeyer S., Tichelaar S. and Steyaert P., FAMIX 2.0 - The FAMOOS Information Exchange Model, technical report, University of Berne, July, 1999, http://www.iam.unibe.ch/~famoos/FAMIX
6. Finnigan P.J., Holt R.C., Kalas I., Kerr S., Kontogiannis K., Müller H.A., Mylopoulos J., Perelgut S.G., Stanley M., Wong K., The software bookshelf, IBM Systems Journal, 36(4), October 1997, pp.564-593.
7. Freijs L.M.G., Krikhaar R.L. and van Ommering R.C., A relational approach to Software Architecture Analysis, *Software Practice & Experience*, 28(3), April 1994, pp. 371-400.
8. Guo G. Y., Atlee J. M. and Kazman R., A Software Architecture Reconstruction Method, *TC2 First Working IFIP Conference on Software Architecture (WICSA 1)*, 22-24 Feb, 1999, San Antonio, Texas, in Software Architecture by P. Donohoe, Kluwer Academic Publisher, pp. 15-33.
9. Holt R. C., Structural Manipulations of Software Architecture using Tarski Relational Algebra, Proceedings of Working Conference on Reverse Engineering WCRE '98, Honolulu, Oct 1998.
11. Hofmeister C., Nord R.L. and Soni D., Describing Software Architecture with UML, *Proc. of the 1st Working IFIP Conference on Software Architecture*, Kluwer Academic Publishers, 1999.

12. Jerding D., Rugaber S., Using visualization for Architectural Localization and Extraction, *Proceedings of the Working Conference on Reverse Engineering, (WCRE97)*, pp. 56-65, October 1997, Amsterdam, Netherlands.
13. Kazman R., Carrière S. J., View Extraction and View Fusion in Architectural Understanding, *Proceedings of the fifth International Conference on Software Reuse (ICSR5)*, pp.290-299, IEEE Computer Society Press, Victoria, B.C, Canada, June 1998.
14. Kazman R., O'Brien L. and Verhoef C., Architecture Reconstruction Guidelines, Carnegie Mellon University, Software Engineering Institute report number CMU/SEI-2001-TR-026.
15. Kazman R., Tool Support for Architecture Analysis and Design, *Joint Proceedings of the SIGSOFT '96 Workshops (ISAW-2)*, ACM, 1996, pp. 94-97.
16. Krikhaar R., Postma A., Sellink A., Stroucken M., Verhoef C., A Two-phase Process for Software Architecture Improvement, Proceedings of the International Conference on Software Maintenance, (ICSM '99), IEEE Computer Society Press, 1999, pp. 371-380.
17. Kruchten P.B., The 4+1 View Model of architecture, IEEE Software, 12(6):42-50, 1995.
18. Lencevicius R., Ran A., Yairi R., ThirdEye – Specification-Based Analysis of Software Execution Traces, *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, Limerick, Ireland, 4-11 June, 2000, page 772.
19. Maccari A. and Riva C., Architectural evolution of legacy product families, Fourth International Workshop on Product Family Engineering PFE-4 Bilbao, Spain, October 3-5, 2001.
20. Murphy G. C. and Notkin D., Reengineering with Relfextion Models: A Case Study, *IEEE Software*, 1997.
21. Ran A., "ARES Conceptual Framework for Software Architecture" in M. Jazayeri, A. Ran, F. van der Linden (eds.), *Software Architecture for Product Families Principles and Practice*, Addison Wesley, 2000.
22. RedHat Source Navigator, http://sources.redhat.com/sourcenav/
23. Richner T., Ducasse S., Recovering high-level views of object-oriented applications from static and dynamic information, *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '99)*, Oxford, 1999, Page(s): 13 –22
24. Rigi: a visual tool for understanding legacy systems, University of Victoria, http://www.rigi.csc.uvic.ca/
25. Riva C., Reverse Architecting: an Industrial Experience Report, *Proceedings. of the 7th Working Conference on Reverse Engineering (WCRE2000)*, Brisbane, Australia, 23-25 November, 2000.
26. Riva C., Reverse Architecting: Suggestions for an Exchange Format, Workshop on Standard Exchange Format (WoSEF), *International Conference on Software Engineering (ICSE 2000)*, June 6, Limerick, Ireland, 2000.
27. Riva C., Xu J. and Maccari A., Architecting and Reverse Architecting in UML, Workshop on Describing Software Architecture with UML, *International Conference on Software Engineering 2001* (ICSE), Toronto, May 2001.
28. Riva C. and Vidal Rodriguez J., Combining Static and Dynamic Views for Architecture Reconstruction, *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering (CSMR 2002)*, IEEE Computer Society Press, 11-13 March 2002 in Budapest, Hungary.
29. Systä T., Static and Dynamic Reverse Engineering Techniques for Java Software Systems, Ph.D Thesis, University of Tampere, Dept. of Computer and Information Sciences, Report A-2000-4, 2000.
30. Turner C. R., Fuggetta A., Lavazza L., Wolf A. L., A conceptual basis for feature engineering. *The Journal of Systems and Software*, 49, Elsevier, 1999.
31. Venice, http://www.cs.Helsinki.FI/group/venice/