AUTOMATIC TRANSFORMATION OF SOFTWARE ARCHITECTURE MODELS

Liliana DOBRICĂ[1], Anca Daniela IONIȚĂ[2], Radu PIETRARU[3], Adriana OLTEANU[4]

Modelul unei arhitecturi software este creat după specificarea cerințelor sau recuperat din codul sursă al sistemului, rafinat pe niveluri succesive de detaliere sau modificat pe acelaşi nivel de abstractizare pentru satisfacerea unor noi cerințe. Rolul arhitectului este realizarea acestor transformări arhitecturale. În lucrare se discută despre tehnicile actuale de transformare automată a modelelor arhitecturilor software. Principala contribuție este identificarea unui cadru de prezentare şi comparare a acestor tehnici, care se deosebesc prin scopul transformării definit de modelul țintă obținut, limbajul de descriere a arhitecturii, şi instrumentul software utilizat în realizarea transformării.

Software architecture model is created from requirements specification or recovered from system code, improved or modified iteratively in its refinement or evolution. The software architect realizes architectural transformations in order to change it. This paper presents an analysis of the current approaches supporting automatic architecture model transformations. The comparison criteria include the goal of transformation, the target architecture model, and the maturity of a tool supporting the techniques. Also we consider other basic modeling concepts such as description based on an architecture description language, views and consistency among views, static and dynamic aspects, functional and quality aspects.

Keywords: software architecture, quality, model transformation, techniques, tools

1. Introduction

Software architecture (SA) is considered of highest importance to the software development life-cycle [20]. It is used to represent and communicate the system structure and behavior to all of its stakeholders with various concerns. Additionally, SA facilitates stakeholders in understanding design decisions and rationale, further promoting reuse and efficient evolution. One of the major issues

[1] Prof., Dept. of Automation Control and Computers, University POLITEHNICA of Bucharest, Romania, e-mail: liliana@aii.pub.ro
[2] Prof., Dept. of Automation Control and Computers, University POLITEHNICA of Bucharest, Romania
[3] Lecturer, Dept. of Automation Control and Computers, University POLITEHNICA of Bucharest, Romania
[4] Lecturer, Dept. of Automation Control and Computers, University POLITEHNICA of Bucharest, Romania

in software systems development today is systematic SA restructuring to accommodate new requirements due to the new market opportunities, technologies, platforms and frameworks. SA transformations require special attention, because of the well-known impact on the project success. Arguments that support this statement can be mentioned. Firstly, SA transformations may be oriented to an evolution changing the source model into a target model and staying at the same level of abstraction. These directly influence the final system properties. Secondly, transformational approaches may be carried out in service of refinement going from a high level SA description to a more detailed one, thus constructing iteratively the final SA, which represents the input of the next development stage. Finally, because of their influence on software quality, they can provide good mechanisms for early-stage quality management. The control of the quality moves to the stage of architectural transformations decreasing in this way production costs and speeding up the time-to market. On the other hand, it enhances the role of the software architect. The architect must be creative in reasoning tradeoffs among different alternatives and applies SA transformations based on his tacit architectural knowledge. SA transformations are not easy to apply and various business drivers (time, resources, costs, etc.) contribute to the complexity of the problem. Automation is desirable, because the manual tasks require not only vast tacit knowledge, but are laborious and therefore cost-intensive, and are error-prone due to the complex design space for human beings.

This paper presents an analysis of the current techniques supporting automatic SA model transformations. This is a very important and poorly understood area of SA, much in need of systematization. Having concrete ways to compare and contrast different approaches would benefit the SA community. Our contribution is the definition of the comparison framework and the presentation of existing approaches based on this framework. The framework includes the goal of transformation, the maturity of a tool supporting the techniques and the basic modeling concepts such as the architecture description language (ADL), views and consistency among views, static and dynamic aspects, functional and quality aspects. Our study distinguishes between SA approaches that are carried out in service of refinement, going from a high level SA model to a more detailed one, and those oriented to SA model evolution, staying at the same level of abstraction. The paper provides solid principles for evaluating SA transformations and it points out important areas that are in need of further research.

2. Background

2.1. Software architecture definition and description

There is not today a clear consensus on a definition of SA or to understanding what constitutes an ADL. During the last decade in the literature

hundreds of definitions have been introduced; several have been cataloged by the Software Engineering Institute (SEI) and are available on the Web [24]. In 2000 an early study identified many alternative notions of what constituted SA and what made up an ADL [25]. Based on a broad survey it has been stated that ADLs capture aspects of software design centered around a system's components, connectors, and configuration. Recently a newer definition of a software system's architecture was given in [26] that is the set of principal design decisions about the system. Design decisions encompass every aspect of the system under development, including design decisions related to: (1) system structure (2) behavior (also referred to as functional) (3) interaction (4) the system's non-functional properties, such as dependability (5) the system's development itself, for example, the process that will be used to develop and evolve the system. It can also be derived definitions for SA models, ADLs, and the act of modeling. An SA model is a document that captures some or all of the design decisions that make up a system's SA. SA models are referred to as architecture descriptions. A model means a formal specification, where a formal specification expects either textual or graphical language with strictly defined syntax and semantics. An ADL is a notation in which SA models can be expressed. SA modeling is the effort to capture and document the design decisions.

In architecture modeling no single set of modeling notations is sufficient for a project. However an architect can choose between a general-purpose notation, such as Unified Modeling Language 2.0. (UML), which is among the richest composite notations or a domain-specific notation, which could be more expressive or highly optimized. A general-purpose notation attracts more users, and therefore will likely be better validated, have more tool support from vendors, and have increased utility as a communication medium among stakeholders. An important property of modeling notations is extensibility (i.e. UML profiles). Extensible notations provide a basic, general purpose foundation for architectural modeling along with mechanisms that allow stakeholders to specialize the notation for their particular business needs, domain, and technology. The main power of a notation comes not through its syntax or even its semantics, but the tools that can be used to operate on the notation. ADLs are supported by a variety of software tools and environments, mainly for editing, visualization, analysis, creating extensions. A good tool support is a driving force behind the widespread adoption of a modeling notation. Conversely, lack of good tool support can leave an otherwise excellent ADL to obscurity.

2.2. Software architecture life-cycle

SA modeling is performed considering various stages of a software architecture lifecycle. Hofmeister et al. have proposed a general model of SA

lifecycle [21]. This model consisted of three stages: architectural analysis, architectural synthesis, and architectural evaluation. This model has been extended to include two more stages, implementation and maintenance (Fig. 1) [22]. All stages are supported by architectural knowledge (AK). AK is divided into four categories [22]: (1) context knowledge, which is a collection of information about the problem space, for instance, architectural significant requirements and the context of a project; (2) general knowledge, which is a collection of knowledge that helps architects to design software, for example, architectural styles and patterns [23]; (3) reasoning knowledge, which is a collection of reasoning information about a design, for example design decisions, design rationale, design alternatives, and trade-offs; (4) design knowledge, which is a collection of system designs such as components and architectural models.

The architectural analysis stage serves to define the problems an architect must solve. An architect examines architectural concerns and context in order to come up with a set of architecturally significant requirements. During the architectural synthesis stage, the architect designs SA solutions for a set of architecturally significant requirements. This task requires an architect to create the proposed solutions. For this purpose, the architect can apply existing solutions (e.g. styles, patterns) to solve the problems at hand. The design is created and synthesized by the architect to capture the design knowledge. The architect also produces the necessary traces between reasoning knowledge, design knowledge, general and context knowledge. Architectural evaluation ensures that the proposed architectural solutions are the right ones. The candidate architectural solutions are evaluated against the architecturally significant requirements. At this stage, an architect shares architecture knowledge with architecture evaluators. This allows the evaluators to learn, search/retrieve, and evaluate the reasoning knowledge and design knowledge. In order to perform an architecture evaluation, they often need to trace reasoning knowledge to context knowledge (i.e. the requirements), general knowledge and design knowledge. When an architecture design is evaluated and approved, architects and reviewers may distill the design as a general design pattern in general knowledge for future reuse.

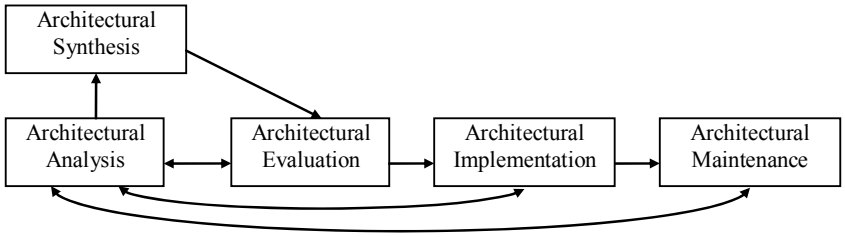

Fig. 1. Software architecture lifecycle

After architecture evaluation, the SA is realized by designers during architectural implementation. At this stage designers and developers need to learn, and search/retrieve the available reasoning knowledge in order to understand the architecture design for implementation. Architects share the knowledge with the implementers to facilitate their understanding. Once the initial system is deployed, architectural changes may take place during the architectural maintenance stage. At this stage, tracing the design knowledge aims to learn about design reasoning and evaluate the impact of certain architectural changes.

3. SA model transformation automation

3.1. SA Model Transformation

The first ideas regarding software architectural transformation appeared in the '90s on the migration trend from code towards software architecture technology. Several definitions of architectural transformation can be found in the literature. Kikhaar [16] defines architectural transformations as operations performed at the code level. Changes applied to the architectural model of a software system are qualified to impact analysis phase and they are left to the software architect experience. Carriere, Woods and Kazman [19] discuss about architectural transformations, too. They describe architectural elements in terms of their static and dynamic features and define transformations in terms of features modification. Early architectural changes are categorized to transformations for understanding, analysis, and modification [18]. The idea towards automatic model synchronization from model transformations has been introduced in [15].

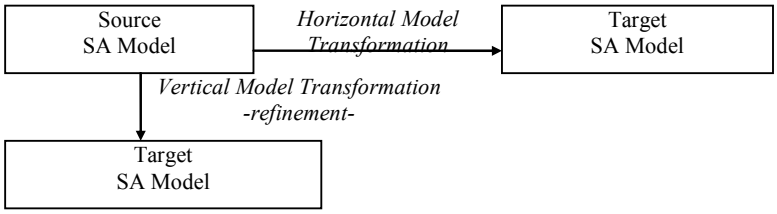

Fig. 2. Model Transformation

Later on model driven development technologies discuss about the idea to automate the process of creating new SA models and to facilitate evolution in a rapidly changing environment by using model transformations [27, 28]. The systematic use of models and reuse of model transformations simplifies and formalize various activities and tasks that comprise the SA lifecycle. We distinguish horizontal and vertical SA model transformations (Fig. 2). In vertical transformations models from higher level of abstraction are transformed to models of lower level of abstraction, e.g. platform independent models to platform

specific models [5]. Here knowledge of platforms is encoded into automatic model transformations, reused for many systems rather than redesigned for each new system. An automatic model transformation specifies how an output model is constructed based on the elements of an input model. Horizontal model transformations are used for describing mappings between models of the same abstraction level. By relating concepts of various types, knowledge of modeling domains is encoded into transformations, enabling the integrated use of models without having to specify relationships between each set models manually.

3.2. Arguments for SA model transformation automation

Model transformation languages aim at automating the process of deriving one model from another one. Thus, when the mapping between two different kinds of models is known, model transformations can provide the following benefits: (1) Repetitive, laborious and error-prone tasks, required to create a model from another model are avoided, as transformations are executed by a tool. (2) Architectural knowledge can be encapsulated in model transformations, ensuring target model quality. (3) The mapping process encapsulated in a model transformation can be easily applied, as software architects applying the model transformations do not need to know the details about how the mapping is performed. (4) Changes are less difficult to manage, as they can be done at the corresponding abstraction level and propagated quickly to lower abstraction levels by model transformations. The SA model in the model-driven process would be updated and then the change propagated to design, implementation and deployment models. Nevertheless, most of model transformation languages have difficulties to preserve manual changes made to a model when the model is updated, so this kind of round-trip engineering is still an open research issue. (5) When several transformations, from a source model to different kinds of target models are available, the same source model can be reused.

SA model transformations are not easy to apply. Firstly, the architect has to remember all the constraints on elements and relationships in order to perform a correct improvement. For certain types of transformation that require vast experience he may need additional design knowledge about the static or dynamic aspects of the system. Secondly, architectural decisions may result in several alternatives of SA improvement; the architect is rarely able to decide which modification to choose, before he understands all consequences of applying a certain approach. Architects have almost no assistance in reasoning about changes. Thirdly, in order to satisfy a new requirement more than one transformation need to be applied to modify SA model and an optimal evolution path needs to be developed. Finally, the architect may need to integrate new crosscut concerns (i.e. security [29]) that could affect the consistency of the SA

model due to modifications of all elements affected by that concern. The execution of a transformation causes a reaction in chain where other architectural changes are required. Usually they propagate in the structure altering adjacent views or hierarchical sub-structures stopping just at the lowest level of the model. Because of the multiplicity of applied transformations and their unpredictable consequences, the process of SA modification is error-prone due to the overwhelmingly complex design space for human beings and time consuming, especially when manually performed by an architect, whose skills to control changes are limited to the ability of remembering a transformation sequence, constraints, or conditions. It is therefore necessary to provide automation tools and techniques to the architectural model transformations.

4. Approaches supporting automatic transformation

This section presents five approaches, which are pattern-based refactoring, sequence of transformations with multiple views extraction, an architecture evolution style, architecture refactoring to improve quality attributes, and evolutionary optimization of an SA model. The presentation framework focus is on the approach description, the goal of transformation, the ADL, the multiple views consistency and the tools to be used in transformation.

4.1. Pattern-based refactoring

Description. Pattern-based refactoring represents the process of transforming a model using a design pattern [4]. This technique is achieved by developing metamodels called transformation specifications that characterize families of transformations. Fig. 3 gives an overview of the main concepts involved in this model transformation approach.

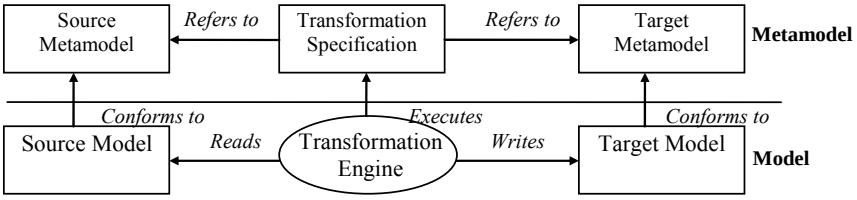

Fig. 3. Metamodelling approach to pattern-based refactoring

A metamodeling consists in patterns specification and transformation rules. Pattern specifications include the problem specification, which is a precise specification of the family of design problems that the pattern addresses; solution specification, which is a precise specification of the designs representing solutions

of the pattern and transformation specification, which is a specification of problem-to-solution transformations defining a transformation language. Composing two or more design patterns could lead to conflicts that must be resolved involving possible trade-off analysis. A validation step is required for models that contain composed patterns.

ADL. A general purpose ADL is considered by pattern-based refactoring approach. Thus it has been applied to SA models represented in UML notation.

Goal. The main goal of automating the process of applying pattern based transformations is to reduce the effort of consistently and correctly realizing a general knowledge that is collected in specific patterns across a SA model.

Multiple views. This approach does not consider multiple views. Functionality conformance is the only concern.

Tools. The software tools are called pattern-aware, embedding codified knowledge of patterns that can be accessed during usage that tools. Pattern-aware tools present patterns as abstraction units that architects can use to construct SA models. A tool support for such approach should provide two interfaces, one for a pattern engineer to evolve and manipulate the tool's representation of the UML metamodel, and the other for the architect to create, manipulate and evolve UML SA models using patterns. Such tools can help in establishing conformance of models to the specification, due to preserving functional properties when defining common properties to problem and solution specifications.

4.2. Sequence of transformation with multiple views extraction

Description. Transformation in SA models is described using a precise mathematical semantics, which is called category theory in [6]. This approach separates computations of a system from its coordination and configuration, allowing the introduction of a dynamic configuration step. SA models are diagrams in the sense of category theory [17] involving explicit superposition and refinement relationships between architectural components. SA is defined by the space of all possible configurations that can result from a certain starting configuration. From this starting configuration, a dynamic step produces the derivation from one SA model to another in a sequence of transformations.

ADL. This technique is expressed by using COMMUNITY, which is a domain specific ADL.

Goal. There are two goals to be considered for SA model transformation. A first goal is to produce SA model derivation in a sequence of transformations as it has been described above. Another goal is to extract multiple views from an ADL metamodel in a systematic way, by listing the design questions each view should answer. Each one of the view types is defined by a metamodel, which is obtained from the architectural metamodel by adding the necessary new entities

and associations. The view's metamodel also show (through a class diagram and OCL expressions) how the new entities are related to those of the SA model.

Multiple views. Multiple views are homogeneous, coherent, relevant, and explicitly related, because they stem from the constructs of an ADL suitable for the description of important architectural concepts. Architectural concepts, their relationships, and their aggregations into various different views are explicitly defined through a metamodel that enables to relate the various views explicitly and enforce their mutual consistency through constraints. Each view can be described in a declarative way through the metamodel, and operationally as a transformation from the architecture. The decisions on which views to define and how to define them is guided by an explicit enumeration of the design questions the architect would like the views to answer.

Tools. There is a workbench developed as a proof of concept This workbench provides a graphical integrated development environment to write, run, debug components and draw configurations of components and connectors. The workbench is extended to provide support for computation, coordination and distribution views.

4.3. Specifying an architecture evolution style

Description. A sequence of transformations is also considered by Garlan in [2] where an architecture evolution style is defined and the possibility to automatically generate possible paths is envisioned. The key is that at an architectural level many systems evolutions follow certain common paths. Each path defines a sequence of SA models in which the first element of the path is the SA model of the current system, and the final element is a desired target SA model. Links between successive nodes in a path are associated with transitions that are composed using a set of evolution operators for that style. In this respect an evolution style is like a state machine for which an execution trace defines an evolution path. Path constraints are specified to constrain the space of paths and to give the correctness dimension of this approach. The evaluation function is introduced for comparison of different paths with respect to quality metrics.

Goal. The goal is to provide automated assistance for expressing architectural evolution and for reasoning about the correctness and quality of evolution paths to achieve business concerns of stakeholders by choosing an optimal path. This asistance is provided by taking advantage of regularity in the space of common architectural evolutions.

ADL. The ADL notation for SA models representation is Acme. SA model is a graph in which nodes represent components and edges represent connectors. Ports are defined as interfaces of components. Annotations with properties of these elements provide more-detailed semantics to represent

reliability (for components), protocols of interaction (for connectors), or signatures of required and provided services (for ports). In this way a list of properties may vary from an SA model to another SA model.

Multiple views. A particularity of this approach is the set of architectures, which is an architectural style and is defined by specifying a vocabulary of architectural structures as a set of component, connector, and port types, together with a set of constraints. Other specifications refer to evolution path properties, path constraints, evolution operators, and evaluation functions.

Tools. This approach has been implemented in a tool called Ævol [2].

4.4. Architecture refactoring to improve quality attributes

Description. Mapping architectural specifications to hypergraphs, then using these to define architecture refactorings is another technique that could be applied automatically [3]. Refactorings are formally specified and a mechanism must be provided to automatically apply them.

Goal. The goal is to preserve architectural behavior and to improve the quality attributes of the architecture. Thus it reduces the development cost and improves the quality of the final system because an automated and systematic search will identify more and better design alternatives. When the architect has to deal with a large number of quality attributes such as safety, availability, reliability, maintainability that conflict with one another and with economic constraints, architecture trade-off analysis methods are appropriate to evaluate design decisions and design alternatives.

ADL. AADL (Architecture Analysis and Description Language) [8] is the underlying ADL in this approach. AADL has been designed on the foundation of MetaH [9]. The goal of AADL is to specifically support model-based quality analysis (e.g. safety with a specific Error Annex [8, 10]) and specification of software and system architectures for complex embedded systems. Architecture specifications are defined as graph-based structures. Graph transformations are identified as a suitable formalism for refactorings. Graph transformations represent the set of architectural design alternatives that are evaluated using evolutionary algorithms and multi-objective optimization strategies.

Multiple views. Only deployment view is considered.

Tools. There is a tool called ArcheOpterix [7] that implements this approach.

4.5. Evolutionary optimization based on metaheuristic search

Description. This approach encodes the challenge of improving SA models as an optimization problem [1]. Metaheuristic search techniques [11] (e.g., genetic algorithms, simulated annealing, etc.) are used to find better SA models.

Goal. The goal of transformation is to automatically improve a given SA model with respect to performance, reliability, and cost.

ADL. The approach is best suited for component-based SAs. Components encapsulate functionality that can be independently reused, and thus component-based SAs provide degrees of freedom to be exploited. In particular, SAs models are expressed with the Palladio Component Model (PCM). PCM strictly separates parametrized component performance models from the composition models and resource models, and it provides configuration options of the models [12]. Quality prediction is done using Layered Queueing Networks (LQN) [13] (or SimuCom EQNs [12]) for performance metrics, Markov models for reliability metrics [14], and a newly introduced PCM cost extension for cost.

Multiple views. This approach does not consider the problem of consistency between multiple views. A view of interest is annotated, then is translated into an analysis model.

Tools. This approach has been implemented in the PerOpteryx tool.

5. Principles for evaluating automatic architectural transformations

The purpose of this section is to offer guidelines related to the selection of the most suitable technique for an automated SA model transformation during SA life cycle. The comparison is based on the framework of the presentation and the focus is mainly on three elements 1) the goal of transformation, 2) the ADLs and multiple view-based SA modeling and 3) the existent tools supporting transformation.

Goal of transformation. The goal of automating the process of transformation could be to reduce the effort of consistently and correctly realizing patterns across a design, to produce derivation in a sequence of transformations, to extract views from an ADL metamodel, to provide automated assistance for expressing architectural evolution, and for reasoning about the correctness and quality of evolution paths to achieve business objectives of an organization by choosing an optimal path, to preserve architectural behavior and to improve the quality attributes of the architecture, to automatically improve a given architecture model with respect to performance, reliability, and cost.

The general problem with quality and software architecture is rooted in the nature of the former. Quality refers to the whole software and thus they cannot be presented in software architecture as components or functions offered by the system, as it is the case with functional requirements. Currently there are approaches that explicitly represent quality requirements in specific models [13][14]. Also software architecture and quality are closely related and they are analyzed together during architectural automatic transformation. Thus quality driven model architecture transformation may be performed automatically.

Architectural description language. According to the level of detail for an SA model description these approaches can be classified into three groups: highest level transformations, which are applied on elements of a deployment diagram [1]; middle level transformations performed on component diagram [2]; lowest level transformations aimed at design patterns and their compositions [4]. Description language is a key issue in the SA automatic transformation. It is impossible to provide any architectural change without adopting a formal architecture representation. Additionally, the complexity of a software structure, the number of viewpoints from which software architecture can be observed, and the great majority of available approaches which can be applied to model and transform architecture result in many alternative description languages like ACME, AADL, UML. 2.0, PCM and other specific quality models. Almost every ADL concentrates on some particular aspects of SA and it is not easy to find a language that can represent all architectural perspectives, from static abstraction levels to system behavior and architectural styles. Architectural transformations cannot be defined before all nuances of the SA are well described in a unified and formalized manner, mainly because changing operations, especially their pre- and post-conditions, must be expressed on the base of established architectural description, to ensure that the system structure is changed in a controlled manner.

UML is strongly related to ADLs and architectural transformations. UML is more popular than any ADL and is used in model driven development with related OCL and QVT languages. Performing or presenting the results of architectural transformations in UML would make them comprehensible to everyone, not only to the specialists acquainted with a specific ADL.

Tools. All the approaches described above supporting automatic architecture model transformation have been included in specific tools or in integrated development environments. Some tools are just workbenches for the proof of concept.

6. Conclusions and further research

This paper discussed about current techniques for supporting automatic architecture model transformations. Automation in architectural transformations depends on the formality and the completeness of the architectural model. A more formal notation is more easily to automation than a less formal one. Similarly, a model that captures a great number of architectural design decisions for the given system will be more agreeable to rigorous, automated transformation than a model that is missing many of such design decisions. Automation is possible in a design process when this process is well understood.

Most of the techniques have shown how they can be used in experiments and prototype implementations. Their results are most often of a preliminary

nature and the prototype implementations are limited and over-simplified. Also compared to real-world systems, most of the case studies are small and have a very limited problem/solution space. This has the benefit that the results can be validated by calculating and interpreting the results manually. However, it remains to be proven that these approaches can handle complex and convex solution spaces in an acceptable time with an acceptable diversity of solutions. In case of simulations the predictions are limited and their precisions depend on the initial assumptions. However the simulation can serve as a basis for experiments and comparisons with real systems in order to improve the models.

Additionally, the applicability and understandability of SA models and tools by common software architects requires experiments to gain insights about the feasibility of these approaches. For example, a special attention must be paid to what kind of information is supplementary required for annotating models.

An open issue remains the toolsets to support automated generation of design alternatives to cope with run-time quality attributes such as performance or reliability. Our current research work focuses on a tool chain development for functional and quality-driven model transformations for various embedded systems domains.

Acknowledgement

This work was supported by CNCSIS –UEFISCSU, project number PNII – IDEI 1238/2008.

R E F E R E N C E S

[1]. *A. Martens, H. Koziolek, S. Becker, R. Reussner*, "Automatically Improve Software Architecture Models for Performance, Reliability, and Cost using Evolutionary Algorithms", in Proceedings of WOSP/SIPEW 2010, San Francisco Bay Area, USA.

[2]. *D. Garlan, J.M. Barness, B. Schmerl, O. Celiku*, "Evolution Styles: Foundations and Tool Support for Software Architecture Evolution", WICSA 2009.

[3]. *L. Grunske*, "Identifying "good" architectural design alternatives with multi-objective optimization strategies", in *Procs of ICSE 2006*, Shanghai, China.

[4]. *R. France, S. Ghosh, E., Song, D.K, King*, "A metamodelling Approach to Pattern-Based Model Refactoring", in IEEE Software, 2003.

[5]. *M. Matinlassi*, Quality driven software architecture model transformation. Towards automation, VTT Publications 608, 2006.

[6]. *C. Oliveira, M.Wermelinger*, "A model driven approach to extract views from an architectural description language", in Procs of. WICSA 2007.

[7]. *A. Aleti, S. Bjornander S., L. Grunske, I. Meedennya*, "ArcheOpterix: An extendable tool for architecture optimization of AADL models", in Proceedings of Mompes 2009.

[8]. *P.H. Feiler, D.P. Gluch, J.J. Hudak*, The Architecture Analysis and Design Language (AADL): An Introduction. Technical report, CMU/SEI-2006-TN-011, 2006.

[9]. *P. Binns, M. Englehart, M. Jackson, and S. Vestal*, "Domain specific software architectures for guidance, navigation and control", in International Journal of Software Engineering and Knowledge Engineering, 6(2):201–227, 1996.

[10]. *L. Grunske, J. Han*, "A comparative study into architecture-based safety evaluation methodologies using AADL's error annex and failure propagation models", in 11th IEEE High Assurance Systems Eng. Symp., HASE 2008, 283–292. IEEE Computer Society
[11]. *C. Blum, A. Roli*, " Metaheuristics in combinatorial optimization: Overview and conceptual comparison", in ACM Computing Surveys, 35(3):268-308, 2003
[12]. *S. Becker, H. Koziolek, and R. Reussner*, "The Palladio component model for model-driven performance prediction", in J. of Systems and Software, 82:3-22, 2009
[13]. *G. Franks, T. Omari, C.M. Woodside, O. Das, and S. Derisavi*, "Enhanced modeling and solution of layered queueing networks", in IEEE Trans. Software Eng, 35(2):148-161, 2009
[14]. *H. Koziolek, F. Brosch*, "Parameter dependencies for component reliability specifications", in Proc. of Workshop on Formal Engineering approaches to Software Components and Architectures. Elsevier, 2009
[15]. *Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Tacheichi, H. Mei*, "Towards Automatic Model Synchronization from Model Transformations", in Procs. of ASE 07, 164-173, 2007
[16]. *R. Krikhaar, A. Postma, A. Sellink, M. Stroucken, C. Verhoef*, "A Two-phase Process for Software Architecture Improvement", in Proceedings of International Conference on Software Maintenance 1999, Oxford, UK, September 1999
[17]. *J.L. Fiadeiro,* Categories for Software Engineering, Springer, 2004
[18]. *L. Dobrica, E. Niemela,* "A survey on software architecture analysis methods", in IEEE Transactions on Software Engineering, vol. 28, no. 7, pg 628-653, 2002.
[19]. *S.J. Carriere, S. Woods, R. Kazman*, "Software Architecture Transformation", Proc. of the Conf. on Reverse Engineering, October 1999
[20]. *L. Bass, P. Clements, R. Kazman,* Software Architecture in Practice, Addison Wesley, Boston, 2003
[21]. *A.M. Babar, I. Gorton, D.R. Jeffery,* "Capturing and using software architecture knowledge for architecture-based software development", in Proceedings of the Quality Software International Conference (QSIC '05), pp. 169–176, 2005
[22]. *A. Tang, P. Avgeriou, A. Jansen, R. Capilla, M.A. Babar*, "A comparative study of architecture knowledge management tools", J. of Syst. and Software, 83 (2010), pp. 352-370
[23]. *L. Dobrica*, "Integrating reusable concepts into reference architecture design of complex embedded systems", Procs. of the 6th Int. Conf. on Informatics in Control, Automation and Robotics (ICINCO 2009), vol. 3, pg. 234-237, 2009
[24]. *** Carnegie Mellon University. How Do You Define Software Architecture? <http://www.sei.cmu.edu/architecture/definitions.html>, Software Eng. Institute, 2005
[25]. *N. Medvidovic, R.N. Taylor*, "A Classification and Comparison Framework for Software Architecture Description Languages", IEEE Transactions on Soft. Eng. 26 (1), 70–93
[26]. *N. Medvidovic, E.M. Dashofy, R.N. Taylor*, "Moving Architectural Description from under the technology lampost", Journal of Information and Software Technology, 2007
[27]. *I. Reinhartz-Berger*, "Towards automation of domain modeling", Journal of Data and Knowledge Engineering, 69 (2010), 491-515
[28]. *A. Olteanu, A.D. Ionita, T. Ionescu*, "Leveraging Open Source ELearning Systems with Web 2.0 and Knowledge Structures", U.P.B Scientific Bulletin- Series C; Electrical Engineering and Computers Science, no.2, (2010), 3-16
[29]. *L. Dobrica, R. Pietraru*, "Security Analysis at Architectural Level in Embedded Software Development", in Control and Applied Informatics, vol. 11, no. 2, pg. 51-58, 2009.