A profile-based approach for maintaining software architecture: an industrial experience report

Claudio Riva¹, Petri Selonen¹,², Tarja Systä²,∗,† and Jianli Xu¹

¹Nokia Research Center, PO Box 1000, FI-33721 Tampere, Finland
²Tampere University of Technology, PO Box 553, FI-33101 Tampere, Finland

SUMMARY
This paper presents our experiences in building a UML-based approach for maintaining software products of a large-scale industrial product family. It enables software architects to define rules and constraints for a product family architecture to be enforced on individual product architectures. The target system of our study was the Nokia ISA platform for a mobile phone product family, a complex software system comprising thousands of components and several million lines of code. We outline our approach and the accompanying tools, and report our experiences and lessons learned in assessing the architectural integrity of 10 ISA platform releases and the associated products.

Copyright © 2009 John Wiley & Sons, Ltd.

Received 10 July 2008; Revised 20 April 2009; Accepted 15 June 2009

KEY WORDS: software product-lines; software architectures; software maintenance

1. INTRODUCTION
Architectural integrity is a key success factor in maintaining the software products of a product family. Ensuring that the individual products conform to the architecture of the product family is, however, not very well understood or supported by the existing maintenance tools and techniques. The evident need for developing such support in the industry is motivated by concrete needs that imply several requirements. First, the approach has to be scalable for large-scale systems, especially when focusing on product families. Second, different kinds of views on the subject systems are desired by the different stakeholders. Third, flexibility that allows the approach to be applied also to other types of systems and domains is preferred.

We have developed an approach and tools that enable software architects to describe the rules and constraints of the product family architecture and to enforce them on the architectures of individual products. The main goal of our approach is to help the software architects to ensure that the software platform has adequate quality, and to have a clear view of the problems while the system is evolving.

The tool environment consists of reverse engineering tools used for architecture recovery as described by Riva [1], and tools for validating the recovered architecture against pre-defined architectural rules as described by Selonen [2]. The first version of the integrated approach and
the tool environment has been earlier reported by Riva et al. [3, 4]. In this paper we provide an overview of our technique and describe how Nokia architects applied the approach to assess the architectural integrity of a real-life product platform and the products built on top of this platform.

Our target system is the ISA platform for the S40 mobile phone product family of Nokia. During the last 10 years, the ISA software architecture has evolved from a small system with hundreds of components and less than 1 million lines of code to a large and complex system with thousands of components and several million lines of code. There are more than 30 different product variants built on top of this platform each year, ranging from middle- to high-range phones with rich multimedia and connectivity features. Four years ago, after a successful small-scale case study, we started to test and improve our approach with the code base and the architecture models of this product family. More than 10 releases developed within a two-year period were analyzed before the approach and tools were transferred successfully to the software architects of product development business units of Nokia.

Ensuring that the architecture descriptions are followed during the implementation of the individual S40 products, or later during their maintenance and evolution, is essential but challenging. Architectural violations may scar the product architecture and make further maintenance activities increasingly difficult. Therefore, controlled software architecture evolution calls for support to create and maintain a clear understanding of the logical components and their connections in the platform and product architectures. To maintain the integrity of the product platform and product architectures, we need support to keep the architecture descriptions consistent and synchronized with the implementation during the evolution of the product family.

We assume that each product family has three major architecture descriptions. Product family architecture is a representative of a set of products, describing their common architecturally significant requirements, architectural rules, patterns, component types, communication infrastructure, and runtime issues. Platform architecture describes the common software assets that are shared among the products and their variation points. Product architecture describes the software architecture of one single product or a set of related products.

2. BACKGROUND

Four years after the first ISA product hit the market, we had two separate research projects, ART and MARVEL, consulting the ISA architects. ART was aiming at developing techniques and tools to guarantee the integrity and consistency of the ever-evolving ISA architecture design, while MARVEL mainly focused on developing approach and tools for reconstructing the implemented ISA architecture structure from the ever-changing and growing code.

2.1. ART: managing variety and inconsistency of architecture design artifacts

The ISA architecture was composed of different architecture artifacts, including the reference architecture, product family architecture, and product architecture. The family and product architectures were constructed from subsystem architecture. A team of architects, including a chief architect and an architect for every subsystem, was responsible for creating and maintaining all the architecture designs. The chief architect was the owner of the reference architecture and coordinated the system-level architecture design, while other architects were in charge of the architecture design under their responsibility, such as the domain, subsystem, and layer architectures.

In theory all the architecture design artifacts should follow the reference architecture, but in reality it was very difficult to achieve this. They were created and updated by different architects at different phases of software development using different design tools, and the architecture designs were even in different forms, i.e., Rational/ROSE UML models, MS Visio UML diagrams, MS PowerPoint pictures, MS Word documents, etc. The entire architecture design suffered from the problems of violating the design, principles defined by the reference architecture, misunderstanding and inconsistency on interface definitions, difficulties in sharing and cross-reviewing the design, and huge effort in updating the design.
The ART project attempted to solve those problems in two steps: first to convert all the architecture design in one form as ROSE UML models, and second to check the integrity and consistency of the UML models of the entire architecture design against the reference architecture. We had two parallel tasks in ART: converting all architecture design artifacts to ROSE UML models, and studying the approach and tool support for architecture conformance checking based on architectural profiles as presented by Selonen and Xu [5].

Architectural profiles were used to capture the rules for product family architecture. They materialize the work context of software architects by defining the structural constraints and rules to which the architecture should conform. They are a variant of Unified Modeling Language (UML) [6] models to support having a common notation for both the reconstructed architectural views and the architectural rules.

2.2. MARVEL: recoding architecture design from code

Owing to the tough competition in the mobile phone industry, time to market has often the highest priority in the product development. As frequently reported by the product managers and developers, there is not enough time to follow a process to make the change from architecture design to code. Many changes are just implemented directly in the code without updating the designs accordingly. When a product is delivered the development of the next one starts immediately, leaving little time for clean-up activities between products.

Before we started the architecture reconstruction task in project MARVEL, many symptoms had indicated that the implementation had gone its own way in a sense that (1) some changes were introduced without considering the integrity of the architecture, architecture rules were violated; (2) some right design decisions were buried or hidden in the code, and they were difficult for the following releases to understand and use; (3) there was simply no clear picture about the architecture as implemented in the code.

The MARVEL project developed an approach for reverse engineering the implemented architecture from the code, called view-based software architecture reconstruction [1]. The main contribution of the approach was utilizing the knowledge of the designed architecture to leverage the abstraction level of code structure (class or function level) to generate the architecture view/model of the implementation (at subsystem and logical component level). MARVEL collected all the information about the architecture models (the components and their dependencies) in a database that was accessible through a web interface. The recovery process was semi-automated with a pipeline of tools in order to repeat the process on any new release of the platform or new product.

2.3. Combining the two approaches

The success of the architecture reconstruction done in MARVEL gave us the chance to compare the implemented architecture and the designed architecture, and also made it possible to use the ART tools for checking the architectural conformance of the implementation model. Once we compared the designed models and the reconstructed implementation model, the difference between them was strikingly obvious. The designed models were severely outdated and the design decisions based on them were questionable. Therefore, the architecture team made the fundamental decision to replace the incomplete human-designed models with the semi-automatically reconstructed architectural models.

We reconstructed the product platform architecture from the actual products to detect possible differences between the actual implementation and the original design rules. Our reverse engineering process was used to recover the architectural model from the implementation of the products into a set of architectural views. The process was based on the analysis of the system artifacts to gather the basic facts about the architecture, and series of transformations that abstracted the facts in the architectural model.

As the reference architecture of the product family and a set of reconstructed architecture model of different releases became the de facto architecture documentation, guaranteeing that the system both evolve into the right direction and its implementation follows, the principles
defined by the reference architecture became a vital issue. Therefore, we decided to combine the architecture conformance checking and architecture reconstruction methods of ART and MARVEL into one unified approach.

The big picture of our intended maintenance approach is illustrated in Figure 1. The reverse engineering process reconstructs a product architecture model for each new product release, guided by the architectural profiles and existing architecture models. The new architecture model, expressed in UML, is then validated against the profiles and architecture models of the previous product releases. The reconstructed architecture model and the validation results are provided to the product architects and designers for analysis, the results of which can be fed back to the next implementation phase.

3. SETTING UP THE APPROACH

To validate whether the platform architecture conforms to a particular architectural style, we started with two obvious things: (1) the software architecture to be studied and (2) the rules the architecture must conform to. For both reconstructing the architecture and capturing the architectural rules, we need to capture the main architectural concepts of the product family architecture.
3.1. Architectural profiles

The original architecture of the ISA product platform was described in writing in a platform reference architecture document with a few complementary UML diagrams. The document described the architecture style used, namely a client–server style, the types of all the architectural elements and their relationships, and the interaction patterns between them. Based on these informal documents, we wanted to describe

1. the components,
2. interfaces, and
3. connectors of the product family.

In more concrete terms, we found that three key architectural concerns were of importance to the ISA architects. These concerns are as follows:

1. validate that only the architectural concepts allowed by the domain are being used, and validate that only allowed relationships between the architectural concepts are being used,
2. validate that the architecture is composed according to the domain (i.e., top-level packages—subpackages—components), and
3. validate that the system conforms to a layered architecture style.

We present the architecture rules using architectural profiles, a variant of UML models to support having a common notation for both the reconstructed architectural views and the architectural rules. This decision grants a few other benefits as well. By using UML, we can capture the rules explicitly in a format that would be both comprehensible for an architect and automatically processable by tools (e.g., using XMI). According to our experience, maintenance approaches end up being strongly dependent on the domain they are applied to. Therefore, we also wanted flexibility with modifying the rules, so that they should not be hard-coded into a particular validation tool. We accomplished this by using UML diagrams for both the architectural views and the rules that the architecture model must follow. Furthermore, this enabled us to store them, manipulate them, and exchange them using standard UML CASE tools.

Architectural profiles capture the rules for product family architecture. They materialize the work context of software architects: they define the structural constraints and rules to which the architecture should conform. Architectural profiles capture and reuse domain-dependent architectural styles and rules that guide both the architecture reconstruction and model analysis.

3.2. Concept determination

We start the architecture reconstruction with concept determination. Concept determination defines the target views for the architecture reconstruction, the architectural concepts, and how they are mapped to the implementation. The target views represent the output of the architecture reconstruction, each emphasizing a particular set of architectural concerns as defined by the profiles.

To enable concept determination, we needed a way of introducing our domain-specific architectural types that are to be recovered. For this, we decided to use UML stereotypes. As defined by OMG [6], a UML stereotype defines ‘how an existing metaclass [defining UML modeling element type] may be extended, and enables the use of platform- or domain-specific terminology or notation in place of, or in addition to, the ones used for the extended metaclass’.

Figure 2 shows examples of defining the concepts for our mobile terminal software system model. The left-hand side of the figure defines four logical component types: Server, Application, Delegate, and CommonApp (represented as extensions of a metaclass Class for tool-related convenience). The right-hand side of the figure defines two logical dependency types: message and invocation. Together, they state the types of components and connectors that can be present in the actual architecture models.

Since we target on producing high-level architecture views, we complemented the traditional bottom–up reverse engineering technique with a top–down approach. First, the bottom–up process extracts source views that are used as a starting point for architecture reconstruction. The source views are then gradually abstracted to target views during the top–down process that is described next. The target views comprise logical components whose types are specified in the architectural profiles.
We used the conceptual profile to identify the logical components and dependencies that are architecturally relevant for the product family and must be recovered from the implementation. We also defined mapping rules between the logical elements and the implementation. The mapping rules represent the traceability links between the logical elements and the implementation, showing how the architectural concepts are mapped to the elements in the implementation. They can also serve to specify what information needs to be gathered from the system artifacts. Ideally, mapping rules are a formal description on how to infer the target view from the implementation. Realistically, the descriptions consist of a set of heuristics and guidelines that will be used during the data gathering activity. This approach allows us to tailor the reconstruction process to the particular architectural style of the system.

3.3. Constraint definition

The existing architectural design documents described a client–server architecture style with the architectural elements and their relationships, together with their interaction patterns. As we already used architectural profiles to define our architectural concepts, we extended the same mechanism to present the constraints as well. To this end, the standard UML profiles are not very intuitive; they are used for specializing existing UML metaclasses with additional constraints, usually expressed using the Object Constraint Language (OCL). UML profiles describe the allowed structures using metalevel concepts, making the profiles obscure and harder to understand for a designer not familiar with the UML metamodel. Therefore, for our architectural profiles we adopted a notation that somewhat resembles to that of the design patterns: the profiles explicitly manifest the allowed structures, making them analogous with the conforming models. The profile notation and its relationship to standard UML profiles are further discussed by Selonen [2].

Our profiles explicitly describe the allowed relationships between architectural concepts: the classifiers, the interfaces, and dependencies and realization relationships between them. The constraints were given using anonymous instance classes: only the stereotypes of the elements in the constraint profiles were taken into account. As a rule of thumb, only the explicitly defined structural relationships in the architectural profiles were allowed in the architectural views, other relationships were interpreted as violations. For example, Figure 3 effectively allows for an element having architectural type ‘Server’ to communicate with an element having architectural type ‘Application’ using a connector of type ‘message’.
3.4. Mapping implementation constructs to architecture concepts

To continue setting up the architecture reconstruction process, we identified how the entities in the conceptual profiles are mapped to the implementation constructs. The logical component types identified in Figure 2 manifest themselves as runtime elements in the ISA platform. The rules for identifying the components from the source code were drawn together with Nokia architects: the source files of a particular component are typically, although not necessarily, located in a single directory, and there is a runtime configuration file for each component. We created a mapping table, mapping approximately 20000 files to logical components, by analyzing the build process and using the domain knowledge provided by the Nokia architects. Creating and validating the mapping tables took a considerable amount of time. We first tried to build scripts to create the mapping table automatically. This took 3 person-months but resulted in a failure due to a large amount of variations. After that we constructed the mapping table manually, which took an extra 2 person-months. Maintenance of the mapping tables, which is currently conducted manually, has appeared to be rather easy.

The conceptual profiles also introduced different connector types. For example, asynchronous messages can be identified from the code based on code patterns that are used to interact with the communications library and the recipient of the message is statically detectable in most of the cases. Function invocations can be identified through observing macros that are provided by the programming language (a variant of C).

3.5. Data gathering

Data gathering was the first activity of the actual view reconstruction phase. The goal was to collect the base facts about the implementation and store them in the source views. The source views contained the relevant information that is necessary for deriving the target views through the mapping rules. The source code is the primary source of information, but other artifacts like make files and configuration files are also exploited. Based on the mapping rules, we analyze the implementation and collect basic facts about the architecture.

The analysis of the source files was completely automated and carried out by a Python script that detects various patterns defined by regular expressions. The output of the analysis was a relational data set that was directly stored in a component inventory. Information about hierarchical organization of the components was prepared together with the Nokia architects and stored in the repository as well. The extracted dependencies between the components were used to derive high-level dependencies between their respective packages. The final component inventory contained the package structure, the dependencies, and a list of all recovered components, accompanied by information about the type, owner, runtime task, and source files.

3.6. Validating architecture views

To give a concrete example on how architecture views are validated against profiles, consider the profile shown in Figure 4. The profile effectively defines our client–server architecture style with message-passing interactions. It explicitly shows the allowed connector types between component types like applications (‘Application’) invoking (‘invocation’) delegate interfaces (‘Delegate IF’) that can be realized (realization) by a delegate application (‘Delegate’), and sending messages (‘message’) to service interfaces (‘Service IF’) that can be realized as common application types (‘CommonApp’).

Figure 5 shows how this profile is used for validating two-example architectural views. The left-hand side of the figure shows examples of both legal and illegal architecture models according to the stereotype and constraint profiles in Figures 2 and 4. The interface elements are abstracted away as is often the case when a component only realizes one interface. The model on the left-hand side contains two illegal dependencies: an ‘invocation’ dependency from ‘Delegate’ component PhoneNumber Delegate App to ‘Application’ component PhoneBook Application, and an ‘invocation’ dependency between two ‘Server’ components PhoneBook Server and UserProfiles Server, neither of which have been defined in the constraint profile. After detecting the
non-conformances, the architecture can be reconciled accordingly. The right-hand side of Figure 5 shows a modified model that conforms to the constraint profile.

After we transformed the reference architecture descriptions into UML architectural profiles, we landed with 6 stereotype definitions and 11 constraint definition parts for conceptual profiles, defining a total of 40 architectural concepts and 68 relationships between them. The architectural concerns our conceptual profiles address are summarized in Table I, together with their UML-level interpretation.

4. ADDITIONAL ARCHITECTURE VIEWS

With the conceptual profiles we could already express how component and connector types are allowed to be connected. However, they did not give sufficient information on how the logical components should be organized and grouped in the target views, nor what the target views are. Thus, we needed something more to guide the architecture reconstruction process.
Table I. Profile types, architectural concerns, and their UML interpretation.

<table>
<thead>
<tr>
<th>Profile type</th>
<th>Architectural concern</th>
<th>UML interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conceptual profile</td>
<td>Validate that only the architectural concepts allowed by the domain are being used; validate that only allowed relationships between the architectural concepts are being used</td>
<td>Check that classes, interfaces, packages, and dependencies have stereotypes defined in the architectural profiles; check that the ‘client-component-type’–‘connector-type’–‘supplier-component-type’ dependencies are defined in the architectural profiles</td>
</tr>
<tr>
<td>Package structure view profile</td>
<td>Validate that the architecture is composed according to the domain (i.e., top-level packages—subpackages—components)</td>
<td>Check that the package containment hierarchy and owned classes follow the parent–child relationships defined in the packaging profiles</td>
</tr>
<tr>
<td>Layer view profile</td>
<td>Validate that the system conforms to a layered architecture style</td>
<td>Check that all the dependencies between classes belonging layers defined in the layering profiles (realized as namespace hierarchies) are directed from a higher-level layer to a lower-level layer</td>
</tr>
</tbody>
</table>

In addition to the component–connector architecture, we wanted to express how the system should be arranged into subsystems, packages, and components. Another, perhaps most important part as far as Nokia architects were concerned, was the ability to enforce layered architecture style. To this end, we introduce two additional view profile types: package structure profile and layer profile. More generally, we want to define a view profile for each selected architectural concern of the architecture descriptions. These view profiles are also summarized in Table I, together with their UML-level interpretation.

4.1. Package structure profile

We use package structure profile to define how the system can be composed as a hierarchy of namespaces. More specifically, we wanted to ensure that the system is composed of ISA Packages, Sub Packages, and Logical Components.

As with conceptual profile, the stereotype definition part on the left-hand side of Figure 6 defines the subsystem types. The constraint definition part of the profile on the right-hand side defines how the system can be composed by using composition associations. ‘ISA Packages’ are the highest-level packages and must contain ‘Sub Packages’. ‘Sub Packages’ can contain other ‘Sub Packages’ or ‘Logical Components’. The multiplicities state that ‘Sub Packages’ must always belong to a particular ‘ISA Package’ or ‘Sub Package’ and ‘Logical Component’ must belong to a particular ‘Sub Package’.

4.2. Layer profile

We used the layer profile to define how the system is to be composed of layers. Again, we started with introducing new architectural concepts, one for each layer. We proceeded to defining how the layers can connect to each other, similarly as with the conceptual profiles.

The stereotype definition part on the left-hand side of Figure 7 introduces the concept of a layer ‘Arch Layer’ and three specific layer types: ‘UI&App Layer’, ‘Service&Resource Layer’, and ‘HW Control Layer’. The constraint profile in the middle of Figure 7 shows what dependencies are allowed between the layers: elements residing at the ‘UI&App Layer’ can only connect to elements at the ‘Service&Resource Layer’, which in turn can only connect to elements at the ‘HW Control Layer’. This effectively describes a three-layer architecture style. The constraint profile on the right-hand side of Figure 7 specifies how a layer can be composed of the architecture concepts defined in other profiles: an ‘Arch Layer’ is composed of one or more ‘Sub Packages’.

We have now introduced new architecture concepts for layers and defined how they can be connected. However, we still need to tie these concepts to the elements in the actual architecture model. Figure 8 shows an example on how the architecture model elements are mapped to the presented concepts. A new ‘Service&Resource Layer’ ISA Service&Resource Layer is tied to ISA Core Services and ISA Protocol Services.
Again, we illustrate how the profile was applied in practice. Consider the left-hand side of Figure 9 that shows an example of a layered view model of an architecture model. There are three layers in the model: ‘UI&App Layer’ Top Layer comprises one ‘Sub Package’ Native Inhouse Applications, ‘Service&Resource Layer’ Middle Layer comprises ‘Sub Package’ Core Services and ‘Sub Package’ Protocol Services, and ‘HW Control Layer’ Bottom Layer comprises ‘Sub Package’ Signal Processing, ‘Sub Package’ HAL and ‘Sub Package’ HW Drivers.

The dependencies crossing the layers should follow the layer view profile. The right-hand side of Figure 9 shows examples of illegal component dependencies violating the profile: both the ‘message’ dependency from ‘Server’ Call Server to ‘Server’ User Profiles Server and ‘invocation’ dependency from ‘Application’ Browser Application to ‘HAL’ LCD HAL violate the architecture rules given in Figure 7.

4.3. Arranging the profiles

Finally, to manage architectural profiles, we arranged them into a hierarchy of UML models as shown in Figure 10. View profiles depend on the conceptual profiles: they import the conceptual profiles and add new concepts specific to a particular view: they emphasize a particular architectural concern of the architecture descriptions.

5. ARCHITECTURE RECONSTRUCTION

After having the profiles in place and concept determination and data gathering complete, we wrapped up our architecture reconstruction with knowledge inference. Knowledge inference generates the target views by abstracting the information of source views to architectural concepts based on the mapping rules between the logical components and the implementation, the structural information from the package structure and layering profiles, and the high-level information from the forward engineering models or previously generated architectural views.
We used the component inventory populated during data gathering to generate various kinds of architectural views. In the context of this work, we were mostly interested in the logical components and relationships. Other interesting views addressed things like task allocation and inter-task communication, organization of the source code and its dependencies, physical location of components in the processing units, runtime implementation of a feature, and organization of the logical components within the development activities like projects, programs or sites. The construction of the views was done by querying the basic data from the repository and by applying view specific rules specified with relational algebra. In the case of the component view, we queried the logical components, packages, and logical dependencies.

The final task of the architecture recovery process was to make the target views available to the architects in an appropriate format like textual reports, graphs, hyper-linked documents, and CASE tool models. In the context of our work, we converted the target views to UML. We mapped the logical packages to UML-stereotyped packages and the logical components to UML-stereotyped classes. The logical dependencies were mapped to UML dependencies.

6. TOOL SUPPORT

Our approach comprises three main toolsets: architecture model analysis and processing toolset, architecture reconstruction toolset, and the model repository and its web interface as shown in Figure 11. The model analysis and processing toolset allows software architects to create the architectural profiles and architectural design models, and check the models against the profiles, while the reverse-architecting toolset provides the latest architectural information from the implementation of various products of the family. The design philosophy of the toolset is based on simplicity, ease of use, and modifiability. The tools are part of a larger software architecting environment reported on by Riva et al. [3].

7. PUTTING THE APPROACH TO PRACTICE

We began to apply our approach with four consecutive releases of the ISA mobile terminal product platform. The tools for model reconstruction and model analysis have been improved and fine-tuned during this first stage in each iteration step of our process. The upper half of Table II shows the sizes of the four reconstructed architecture models. Roughly, the architecture models have around 200 subsystems, 1000 components, and 10 000 connectors.

When observing the figures, one quickly notices that the size of the architecture model seems surprisingly unstable. The number of subsystems was growing considerably, while the number of components reduces at the same time; also, the ratio of connectors per components changes.
between the second and third releases. The reason for this instability is the platform architecture reconstruction process: to come up with the product platform architecture, we chose to reconstruct the architecture models of single products built on top of the ISA platform and merge the resulting models to construct the common product platform.

The lower half of Table II shows the detected architectural incidents relative to the total number of components and connectors: concept non-conformances, connector non-conformances, and layer non-conformances.

The first indicator is the number of concept non-conformances that manifest themselves as usage of illegal (i.e., undefined) UML stereotypes. If the architecture models were forward engineering models manually created by the architects, the incidents would point out actual conflicts in the design. Since our architecture model was reconstructed, the concept non-conformances manifest the quality of the mapping process during reverse engineering: the non-conformances pinpoint components that do not have a proper mapping. As we improved the architecture reconstruction
process after each iteration step, the quality of the mappings improved accordingly and reached a level of maturity. As a result the concept non-conformance was reduced to 0 for the models reconstructed from later releases.

With the connector and layer non-conformances, we made a conscious decision to restrict our efforts on a non-conformance subtype that was considered to be the most important by the Nokia architects: connector non-conformances between clients and servers. After each round of applying our approach, we gave the results to the Nokia architects. The architects evaluated the non-conformances and identified a subset of them that was specifically targeted when maintaining the next product release. The figures are anecdotal by nature, but they do suggest that the number of incidents has gone down in all the three monitored categories.

After this stage we felt confident about our approach and we transferred the technology to the product development department. The approach has been used in the product development and maintenance since then. To summarize the application of our approach, we show the analysis results of 10 releases developed in 2 years period (including the four releases we analyzed previously). Figure 12 shows the changes of the complexity of this software platform. The fluctuation of the complexity is caused by the frequently changing market requirements, adding new product feature configurations, introducing new hardware modules, and sometimes the change of development organizations, etc.

The main goal of our approach is to help the software architects to ensure the adequate quality of the software platform and to have a clear view of the problems while the system is evolving. Figure 13 indicates that our approach has achieved this target. The two architectural incidents, client–server connector non-conformance and layer non-conformance, have been controlled between 1 to 2%, in spite of the platform constantly changing. In reality a software system cannot be 100% compliant to its designed architecture profile due to the factors explained in the following section. Our approach gives a clear picture of what and where those non-conformances are; hence, the architects can further investigate the problems based on the analysis results.

8. ANALYZING THE RESULTS

After each validation round, the results were reviewed by Nokia architects. As expected, many of the non-conformances originated from the quick-and-dirty solutions (e.g., during bug fixing) under the time-to-market pressure. The initial findings from the investigation revealed several sources of the problems both in the implementation and our approach.
Evolving and incomplete profiles. When new stereotypes were introduced, the constraints regarding their dependencies could not be completely specified, and some of them were decidedly left open. These dependencies were reported illegal. On the other hand, some components turned out playing different roles than what had been defined by the stereotype. This usually resulted from incomplete mapping during architecture recovery. However, when this was not the case, the design and implementation of such components should be reviewed.

Inappropriate function allocation. Our investigation revealed a significant source of layering non-conformances caused by a set of top layer components controlling global data. Most of the components in the lower layers depended on them, breaking the layered architecture. As a result of the investigation, these components were migrated accordingly.

Performance and other non-functional requirements. Some of the dependencies involved top layer components directly invoking the functions of the bottom layer components by-passing the middle layer. Such dependencies are sometimes necessary in a real-time and embedded system for improving performance. However, they must be monitored and closely controlled within a very limited scale. The architects would classify such non-conforming dependencies as allowed or premeditated non-conformances.

Sharing of code in the same development team. Several illegal dependencies occurred between components that were implemented by the same team or several closely cooperating groups working on the same feature or a feature set by, for example, sharing a function from another component. Under time-to-market pressure, it became clear that developers physically located close to each other tend to take such shortcuts. Sharing of code in the same development team proved to be an interesting instance of ‘Conway’s Law’ regarding similarities between software structure and organization: ‘any piece of software reflects the organizational structure that produced it.’ When such non-conformances were detected and removed from the next release, they would slowly but surely find their way back to the architecture in the subsequent releases.

9. RELATED WORK

Maintenance and evolution of large-scale software systems is known to be challenging. Besides the complexities that are caused by the size of the software, different stakeholders might have different needs for changes, which pull the evolving software to different directions. Siy and Perry [7] discuss the complexity of evolution of large-scale systems based on their study related to Lucent Technologies 5ESS® large-scale switching system.
Svähnberg and Bosch [8], in turn, focus on the evolution of software product-lines. They studied two Swedish organizations that have employed the product-line architecture approach for several years. While we have proposed a practical architecture recovery and validation approach and discussed its application for maintaining the ISA system, they focus on identification of a categorization of the types of evolution in product-lines, related to the requirements, the software architecture, and the software components.

Dikel et al. [9] discuss the development of large-scale product-lines, based on a study on Nortel’s solution. As a result, they have identified six principles that help in reducing the complexity of an evolving family of products and maintain the integrity of the architecture. Instead of such, without a doubt useful recommendations, we aim at identifying the implications of changes and violations of the architectural rules’ set. The actions to be taken based on our findings is left to Nokia architects and are out of the scope of this paper.

Yet another viewpoint on large-scale software architecture analysis is presented by Lung et al. [10]. They propose an approach to capture and assess software architectures for evolution and reuse purposes. Their approach consists of a framework for modeling various types of relevant information and a set of architectural views for, e.g., analysis purposes. This is comparable to using architectural profiles in our approach. As a part of the support provided for architecture analysis, they also aim at detecting design violations. As in our approach, they compare the ‘as-intended’ and ‘as-built’ software architectures for identifying the violations. They have applied the approach to large-scale telecommunication systems.

10. DISCUSSION

We presented a novel profile-based approach for maintaining software architectures and its application to support maintenance of ISA platform for a mobile phone product family of Nokia. We broke down the architecture recovery process to concept determination, data gathering, knowledge inference, and presentation. The architecture reconstruction approach follows the view-based reconstruction process proposed by Riva [1] and van Deursen et al. [11]. The presented set of architectural profiles and their interpretation has been adopted from the work of Selonen and Xu [5]‡ and Selonen [2].

The approach was set up for and evaluated by analyzing ten consecutive product releases built on the Nokia ISA product platform. The experiment demonstrates that our approach and tools are flexible, robust and they scale up to large systems. The reconstructed architecture views, showing the real, implemented platform architecture, were actually used to replace the original architectural design documents as the official architecture description.

While the architectural types are domain-specific, the component–connector and layered architecture styles used are obviously general. We recognize that setting up the approach—creating the architectural profiles and defining the mappings for architecture reconstruction—is a significant effort; however, as explained in what follows, our experiences and the feedback from Nokia engineers suggest that this effort is well rewarded.

Besides the actual sources for architectural non-conformances, the validation may also result in several false positives. Therefore, a careful analysis of the results is always needed. The accuracy of the results depends on how the architectural profiles are defined, and naturally also on the subject system to be analyzed. The flexibility provided by architectural profiles is essential also for the approach to be applicable for different projects and domains; experiences suggest that maintenance approaches are strongly dependent on the domain, to which they are applied.

We have learnt several lessons from the development of the approach and toolset for maintaining software architectures, and specifically from their applications. First, a tool that is able to recover software architectures automatically, e.g., based on predefined mapping rules, works only for...

‡The applied rules are variants of the stereotype conformance, relationship conformance, interface conformance, and multiplicity conformance rules [5].
software of certain scale and complexity, such as our subject ISA software, discussed in this paper. ISA is a large-scale product platform; the target architecture model has around 150 subsystems, 1000 components, and 15,000 dependencies. Maintaining ultra-large-scale software, however, would require an interactive tool that allows involvement of experience software architects to abstract the architectural information from code-level. This is to be done through several steps of grouping, mapping, and abstraction.

Our second lesson is that a tool that supports automated architecture recovery and validation may lead a development organization to a wrong direction unless the organization has a proper architecture management team and program, the danger being that the tools are used to replace the work that should be done by people: code reviewing, design documentation, architecture documentation, etc. Automating these important tasks and relying on such a tool may lead to tool implementation-driven architecture maintenance activities. Instead, the architecture management should first set up the target architecture (in a form of architecture models, profiles, etc.), and the tool could then be used to measure its distance from the implementation, exposing architectural problems in the implementation. The approach and tools developed and discussed in this paper are, we believe, a step toward that direction.

According to informal feedback from the Nokia architects, our approach gave them valuable information for improving the architecture design and implementation. The maintenance approach makes the architects aware of architectural problems in the implementation and allows them to monitor and improve the quality of the software constantly. We were privileged to have our approach and tools deployed at the ISA product development process in 2004 and transferred to the Nokia software architects.

ACKNOWLEDGEMENTS

The authors wish to thank Nokia, Jani Airaksinen for the implementation of the artDECO tool kit, professors Ilkka Haikala, Kai Koskimies, and Gail Murphy for their useful comments, and the PRACTISE team members for their technical support. The work has been partly funded by Eureka Σ 2023 Programme, under grants of the ITEA projects i04032 (SERIOUS) and i02009 (FAMILIES), and by the Academy of Finland.

REFERENCES


**AUTHORS' BIOGRAPHIES**

Claudio Riva received the MSc degree in Telecommunications Engineering from the Politecnico of Milan, Italy, in 1998 and the PhD degree in Computer Science from the Vienna University of Technology, Austria, in 2004. Between 1998 and 2008 he worked as a researcher and principal scientist in the software laboratory at the Nokia Research Center where he conducted research on software architectures, software engineering, and reverse engineering. Since 2008, he is heading the service creation group in the Nokia Emerging Markets Services. He has published several conference papers about software architectures and software maintenance and, participated in several review committees. In 2004, he was the Program Chair of the European Conference on Software Maintenance and Reengineering (CSMR). He is a member of the IEEE Computer Society.

Petri Selonen is a Principal Member of the Engineering Staff at the Nokia Research Center in Tampere, Finland. Before joining Nokia, he worked as an assistant professor at Tampere University of Technology and as a postdoctoral research fellow at the University of British Columbia. His current research involves Web services engineering, platform development and ReST.

Tarja Systä is a professor at the Tampere University of Technology, Department of Software Systems. Her current research interests include software maintenance and analysis, software architectures, model-driven software development, and development and management of service-oriented systems.

Jianli Xu is a Senior Architect in the R&D unit of Nokia Devices. He was a Principal Scientist at the Nokia Research Center when the project described in the paper had been conducted. He is currently working on system architecture study and system architecture modeling with UML/SysML.