UML-based Reverse Engineering and Model Analysis Approaches for Software Architecture Maintenance

Claudio Riva[1], Petri Selonen[2], Tarja Systä[2], Jianli Xu[1]

[1]Nokia Research Center
Software Technology Laboratory
{claudio.riva | jianli.xu}@nokia.com

[2]Tampere University of Technology
Institute of Software Systems
{petri.selonen | tarja.systa}@tut.fi

ABSTRACT
This paper proposes a UML-based software maintenance process. The process is guided by architectural descriptions and existing architectural models. The descriptions are given as variants of UML profiles describing the styles and rules relevant for a particular application domain. A reverse engineering subprocess, combining top-down and bottom-up reverse engineering activities, aims at constructing the architectural models. Resulting models are investigated in a model analysis subprocess. The models are checked against the profiles to find violations against the given architectural rules when maintaining and developing the subject system, and they are further analyzed using a set of UML model processing operations. The proposed approach is applied for maintaining a large-scale product platform architecture and real-life product-line products built on top of this platform. The model analysis results of the case study are discussed.

Keywords
Reverse Engineering, Software Architecture, Architecture Maintenance, Architecture Analysis, UML

1 INTRODUCTION

The Unified Modeling Language (UML) [OMG02] has established itself in software industry for describing software models. This paper discusses a UML-based software maintenance process relying on two subprocesses: *reverse engineering* and *model analysis.* The former combines a top-down reverse engineering technique with traditional bottom-up reverse engineering activities, while the latter utilizes a set of UML model processing techniques for analyzing the architecture models. During the evolution of the software architecture, the whole procedure is iterated for individual releases of the subject system. The main driver of the presented work has been to enable the software architects to analyze the structural dependencies that exist in large software systems. 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 search to extract "the big picture" of the system can bring up a clear understanding of its major components and their interactions.

To customize UML for a certain domain or for certain (e.g. company-specific) design or architecting policies, UML *profiles* [OMG, Sec. 2.14] can be used. A profile is essentially an extension the UML specification using its built-in extension mechanisms. We define an *architectural profile* [SeX03] as a UML profile variant which presents a set of architectural rules and specifies a subset of UML models that can be considered legal in a particular context. In essence, the architectural profiles describe an architecture modeling language that the designer must follow.

In the reverse engineering subprocess, finding the mappings between the source code and the UML architectural descriptions requires several steps. The subprocess is guided by the architectural profile descriptions constructed for a product line, product family, or product platform under investigation: the profile defines what concepts are architecturally relevant and to be recovered from the implementation. When maintaining software architectures, changes in the source code during the evolution of the software need to be mapped and reflected against the architectural descriptions and rules. These rules and profile descriptions as well as the pre-existing software architecture models can be used to guide the reverse engineering process, especially when constructing abstract and design level UML models from the extracted low-level data. Still, the construction of UML models from the source code is far from straightforward, even if an object-oriented programming language has been used.

Because of the differences in concepts at the design and implementation levels, interpretations are necessary even for the extraction of class diagrams from the source code. Currently available CASE-tools supporting UML-based reverse engineering are rarely able to produce other models besides class diagrams [Kea02]. Consequently, we use a middle model, containing the detailed and accurate information of the subject software, from which the UML views are constructed. This way, the reverse engineer has a better control on the model construction process. The graph-based middle model further allows the composition of alternative views and abstractions.

In the model analysis subprocess, the models are checked against the profiles to indicate whether the given architectural rules have been followed when maintaining and further developing the subject system. Also, various UML model processing operations can be defined and used e.g. to build alternative abstractions and to compare different versions of the software architecture and to find out how it has been evolved. The operations can be used e.g. to compare the original design-level models against the models reverse engineered from the source code. We have developed a technique and a tool for validating a given UML model against the architectural profiles [SeX03]. The architectural profiles should be followed correctly in the architecture design and maintenance process in order to guarantee that the resulting architecture design has necessary properties, or lacks undesirable ones. The used model processing operations have been implemented on top of a CASE-tool independent model processing platform xUMLi [Aea02][PeS04].

This paper introduces the architecture maintenance approach and gives examples of its application for maintaining a large-scale product platform architecture and real-life product-line products built on top of this platform.

2 OVERVIEW OF THE APPROACH

Usually a system under development can be seen at several levels of abstraction: starting from the most abstract level, we can have requirements, architecture models, design models, and finally implementation at the most detailed level. In addition, even new systems built from scratch typically rely on, or are subject to, existing architectures and platform or domain specific rules and restrictions. A system is typically not static but evolving over time due to e.g. changing requirements, changing technologies, bug fixes, adding of features, and re-engineering. To be able to maintain a software system, a necessary level of design documentation must usually be kept up and maintained in synch with the actual implementation model. When there are changes in the system, these changes should propagate through the whole model hierarchy. This is essential to produce a well-maintained model.

With a single system, the synchronization of the model hierarchy can be done by hand, though this is often tedious and error-prone. However, when the focus is switched onto product lines, product families, and product platforms, the importance of the models dramatically increases, as does the effort involved in maintaining them. Changes in product platforms affect all its products depending on the particular architecture. When several products and several design teams are involved, the need to be able to specify domain, product-line, and product platform specific conventions, rules, and restrictions grows more important.

Figure 1 illustrates the overall maintenance process described in this paper. The two enclosed subprocesses are a reverse engineering process (RE-process) and a model analysis process (MA-process). These processes involve the tools, techniques, and also design decisions made by participating designers. The scheme also includes three models. The implementation model, typically represented as (nested) graphs in reverse engineering tools, includes both the source code and more abstract representations in a suitable format. The architectural views are architecture representations and analysis views in UML. The architectural profiles describe the domain-specific concepts and architecture conventions, styles, restrictions, and rules as a set of UML profiles.

The RE-process combines bottom-up and top-down reverse engineering approaches. The existing

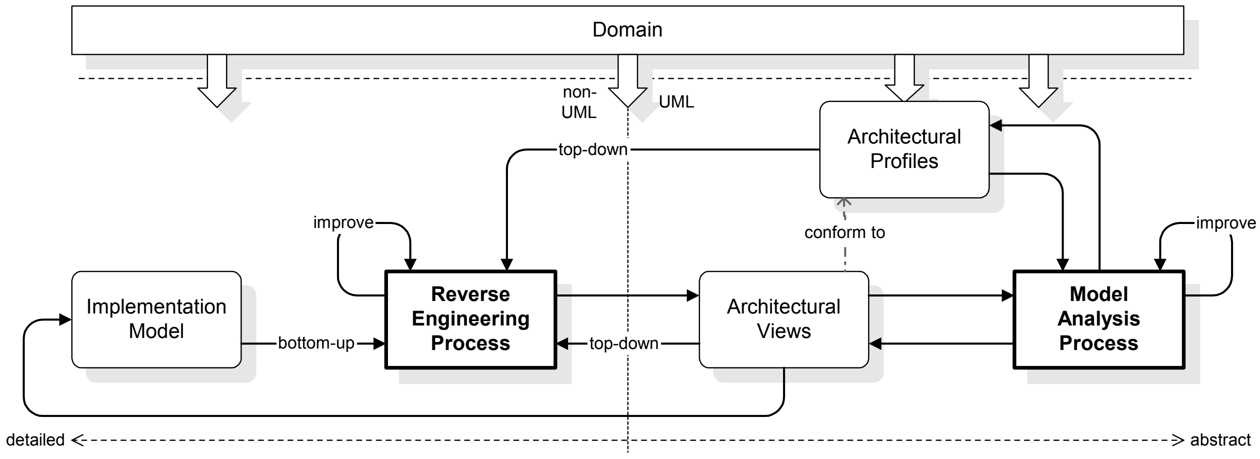

Figure 1. Overview on the maintenance process

architectural views (if any) and architectural profiles guide the subprocess from a top level. The implementation-level artifacts are gradually abstracted and mapped to the high-level concepts. The result is a new (set of) architectural views, translated into UML. The MA-process analyzes these architectural views against architectural view profiles and existing architectural views, either against previous versions or reference models (e.g. forward engineering models, reference architectures).

Both subprocesses build new models based on existing ones. In addition, human interaction is involved throughout the whole maintenance process and affects the included subprocesses. Consequently, the maintenance process is not static but incremental and iterative: new versions of architectural view profiles, architectural views, and implementation models are produced whenever necessary. Both involved processes are self-improving: during each iteration cycle, they are evaluated against the new and existing models and modified accordingly if necessary.

As a typical usage scenario for the process, consider a situation where a (potentially outdated) forward engineered architecture model already exists. An initial description of the domain is mapped to a first version of an architectural profile. The process goes as follows:

1. The RE-process takes the existing forward engineered model, together with the architectural profiles, as its top-down level input. Guided by them, the bottom-up reverse engineering is started. After generating a set of intermediate model representations, each typically raising the level of abstraction, a suitable level of abstraction is reached.
2. The architecture model generated in step 1 is mapped into UML views to be used by the MA-process.
3. The MA-process validates the architecture model generated in step 2 against the architectural view profiles and reports the incidents potentially violating the domain-specific conventions and selected architectural styles and constraints. The MA-process can compare the new set of architectural views against the previously generated ones.
4. The results obtained in step 3 and additional view generation (i.e., merging, slicing, synthesis) are analyzed and necessary steps are taken, either to improve the MA-process, the RE-process, or the architecture model itself.
5. The changes introduced in step 4 are mapped back to the implementation model.
6. The changes in the implementation model, the architectural profiles, and the architecture models are, in turn, used for modifying RE- and MA-processes.

The cycle can begin again from step 1.

The process described in Figure 1 does not in any way dictate in which order the overall process is executed, nor does it take any position on how the subprocesses are constructed: what tools should be used, what the models should be like, etc. Experience suggests that the process, subprocesses, and related techniques should be – and must be – customized for every domain they are applied to. A process execution cycle can modify several models simultaneously. Since the description of the processes can also be seen as models, the models themselves are subject to changes and improvements.

3 APPLYING ARCHITECTURAL PROFILES

Architectural profiles represent the common architecture modeling language shared by both the RE- and MA-processes, usually in the form of a *conceptual profile* and a group of *view profiles* [SeX03]. The conceptual profile defines the fundamental concepts (types) that are used throughout the entire architecture design and description, and the view profiles specify the concepts and rules that are specific only to this architectural view. We define a view profile for each selected architectural concern of the architecture descriptions. Together the architectural profiles are put in a profile hierarchy where all the view profiles depend on the conceptual profile. A view profile may depend on other view profiles as well, depending on its properties.

Next, we present a profile hierarchy, corresponding profiles, and the process of establishing the two. While they are tied to a particular case study, they should go on to illustrate the adopted profile-based approach. The case study, concerning a product-line of mobile terminal software built on top of a common platform, is discussed in more detail Section 6.

The original software architecture of the product platform was described in a platform reference architecture document and an architecture design model. Basic UML notations, mainly class diagrams, were used for describing the reference architecture and the logical view model [RXM01]. The platform reference architecture document is a textual document with some complementary UML diagrams. The reference architecture document describes 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. To use our tools for automating the architectural model validation process, we transformed the reference architecture descriptions into UML architectural profiles.

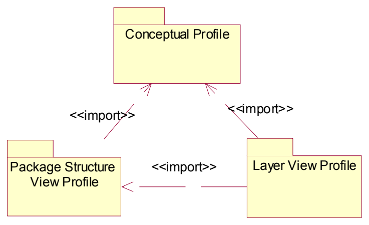

Figure 2. Profile structure

Figure 2 shows the profile structure of the mobile terminal software architecture used in our case study. There are two view profiles: *Package Structure View* profile and *Layer View* profile. All profiles have two parts, a *stereotype definition part* that defines the types of architectural elements with UML stereotypes, and a *constraint part* that specifies the allowed relationship among those types.

Conceptual Profile

The Conceptual Profile describes the conceptual model that specifies the architectural style and the validation rules with class diagrams. Figures 3 and 4 show examples of the stereotype definition for the mobile terminal software system model. Figure 3 defines four logical component (or design components in contrast to implementation components): *Server*, *Application*, *Delegate*, *CommonApp*. Figure 4 defines two logical dependency types: *message* and *invocation*.

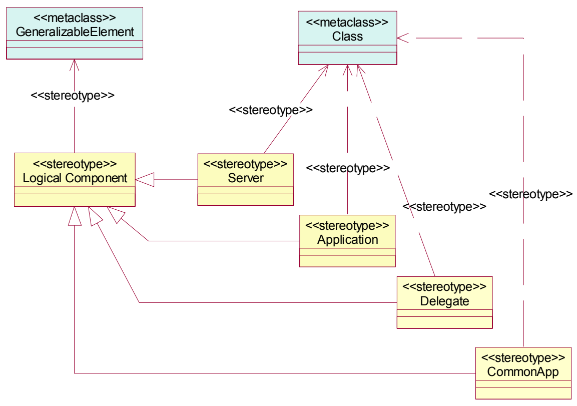

Figure 3. Stereotype definitions for components in the conceptual profile

Figure 5 presents one example of the constraints in the conceptual profile describing the allowed relationships between the architectural concepts. The profile essentially specifies the interfaces, their realizers, and the classifiers depending on them. The constraints of the profile must be satisfied in the architecture model. Figure 5 effectively defines the basic architectural style of the software system: a client-server system with message-passing interactions.

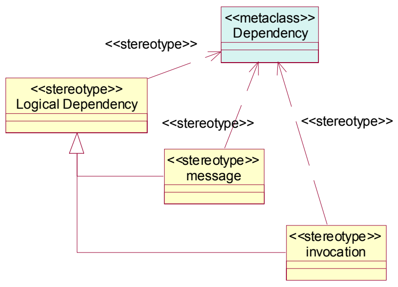

Figure 4. Stereotype definitions for dependencies in the conceptual profile

Package Structure View profile

The Package Structure View profile defines how the system is decomposed into a hierarchy of namespaces (subsystems, packages) and which namespaces are allowed to reside under each other. Figure 6 shows a simplified example of the profile: on the left side the package types are defined through stereotypes and, on the right side we define the constraints of the system decomposition structure. ISA Packages are the highest-level packages and can contain Sub-Packages. Sub-Packages can contain other Sub-Packages, or Logical Components. This package hierarchy mainly describes a functional decomposition of the system, but it also serves as a means of organizing and managing both the design and implementation artifacts.

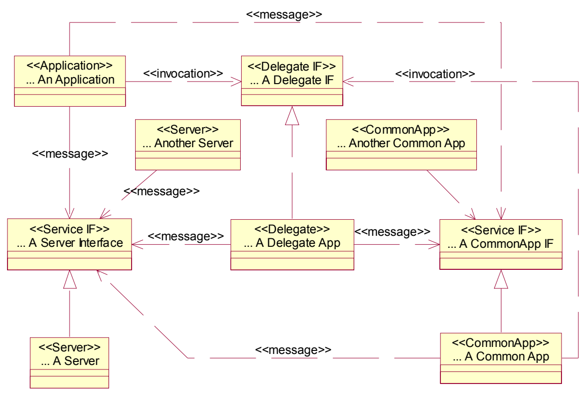
<<message>>
<<Application>> ... An Application
<<invocation>>
<<Delegate IF>> ... A Delegate IF
<<invocation>>
<<Server>> ... Another Server
<<CommonApp>> ... Another Common App
<<message>>
<<message>>
<<Service IF>> ... A Server Interface
<<message>>
<<Delegate>> ... A Delegate App
<<message>>
<<Service IF>> ... A CommonApp IF
<<Server>> ... A Server
<<message>>
<<CommonApp>> ... A Common App

Figure 5. Constraint definition for the conceptual profile.

Layer View Profile

The layer view profile introduces a conceptual decomposition of the system. The stereotype definition part in Figure 7 introduces the concept of a layer and three specific layer types. The constraint profile specifies how to a layer can be composed of the architecture concepts defined in other profiles and what dependencies are allowed between the layers (left-hand and right-hand side of Figure 8, respectively), effectively leading to a three-layer architecture style. After architecture model elements are mapped to the presented concepts, the model can be recomposed and the relationships checked. We can also define other conceptual views by giving alternative system decomposition, for example, to define an organizational view showing the development teams and their relationships based on the components they are in charge of.

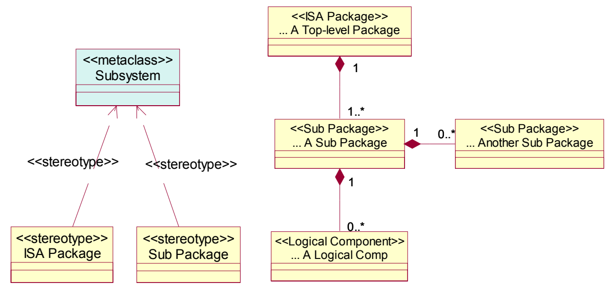

Figure 6. Example of package structure view profile

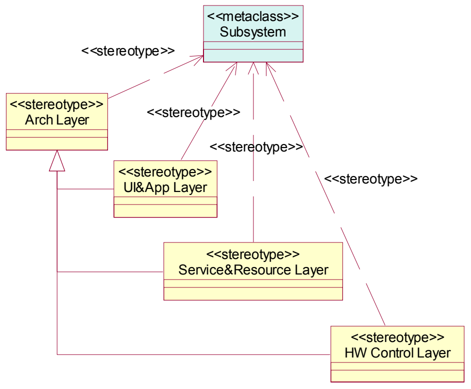

Figure 7. Layer definition in the layer view profile

4 REVERSE ENGINEERING PROCESS

The goal of this activity is to recover the architectural model from the implementation of the products and to construct the architectural views. The process is based on a series of transformations that abstract the facts extracted from the implementation in the UML model. Starting from our previous work [Riv00], we have elaborated a robust architecture reconstruction process based on the selection of the views to recover [Deur04].

The target view of the reconstruction is the *Component View*. The view describes the logical components and their logical dependencies as defined in the Conceptual Profile, and their hierarchical organization as defined in the Package Structure View profile. The component view also calculates the high-level dependencies among the packages. The Conceptual Profile defines what concepts are architecturally relevant for the product family and must be recovered from the implementation. In this way, the reconstruction process is completely driven by the concepts in the profile.

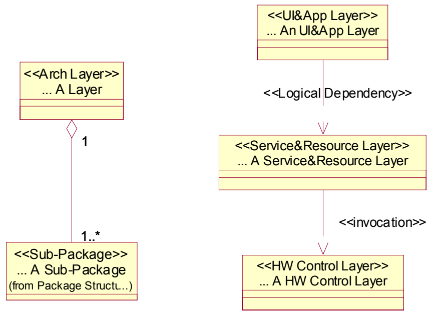

Figure 8. Construction and dependency rules of layers

The process consists of the following tasks: (1) mapping the logical concepts to the implementation, (2) gathering the data into a component inventory, (3) constructing the architectural views, and (4) representing the views in UML. The process is tailored to our particular product family and it can be applied on all the various products.

Mapping the logical concepts to the implementation

The first task is to identify how the entities in the Conceptual Profile are mapped to the implementation. The logical components of Figure 3 are runtime elements and become active they register to the communication infrastructure. Discussions with the architects identified the practical rules for identifying the components in the source code: the source files of the component are typically, but not necessarily, located in one single directory and for each component there is a runtime configuration file. There is no easy way to map the files to the component and, thus, we manually mapped approximately 20.000 files to the logical components. The mapping table was created by analyzing the build process and with the help of the domain knowledge of the architects.

As shown in Figure 4, there are two types of dependencies: asynchronous messages and function invocations. Messages are sent between components through the communication infrastructure. In the source code, we can identify a message passing from particular code patterns that are used to interact with the communication library. The runtime recipient of the message is statically detectable in most of the cases. Function invocations are made through macros provided by the programming language (a variant of C).

Data gathering and the component inventory

The second task is to gather all the relevant information from implementation and to store it in a component inventory (a relational database). The analysis of the source files is completely automated and carried out by a Python script that detects various patterns defined by regular expressions. The output of the analysis is a relational dataset that is directly stored in the database. The information about the hierarchical organization of the components have been prepared together with the architects and stored in the repository. The dependencies are used to calculate the high-level dependencies between the packages and stored in the repository as well. The final component inventory contains a list of all the recovered components, (with information about the type, the owner, the runtime task, the source files), the package structure, and the dependencies. Since our goal is to create an architectural view of the overall platform of the family, we analyze the single products and merge the final results.

Constructing the architectural views

From the component inventory we can generate various architectural views about the family platform: *component view* (the logical components and the logical relationships), *task view* (task allocation and inter-task communication), *development view* (organization of the source code and their dependencies), *deployment view* (physical location of components in the processing units), *feature view* (run-time implementation of a feature), and *organizational view* (organization of the development activities like projects, programs, sites). The construction of the view is done by querying the basic data from the repository and by applying several rules specified with relational algebra. In the case of the component view, we query the logical components, packages and logical dependencies.

The result can be typically visualized as a hierarchical directed graph where the nodes represent packages and components and the arcs represent logical dependencies. We can export the graphs in various formats: hyperlinked pages, SVG, Rigi[1] and other visualization tools.

Transforming the views into UML

The final task of the reverse engineering activity is to transform the component view to the UML language. We map the logical packages to UML stereotyped packages and the logical components to UML stereotyped classes. The logical dependencies are mapped to UML stereotyped dependencies. Figure 9 shows an example of the reverse engineered component view in UML format. It shows the top-level packages and the high-level dependencies.

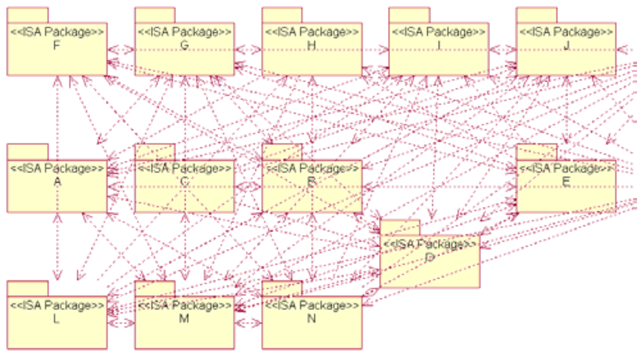

Figure 9. The Package Structure View in UML

5 MODEL ANALYSIS PROCESS

The model analysis process is supported by tools and techniques for analyzing and processing the architecture model produced by the RE-process. In the following, some of the tools and techniques are explained.

Architecture validation against profiles

The UML architectural profiles are used as the mechanism for introducing lightweight extensions to the UML metamodel, describing domain-specific architectural conventions, rules, and constraints. From a practical viewpoint, the profiles can be seen as visual shortcuts representing examples of allowed architectural constructs. Their interpretation is given by evaluating them in *a posteriori* fashion by a selected set of *conformance rules*. Together, the set of profiles and the rule configuration define an architecture language. The architecture model is validated against these profiles and the found non-conformant incidents are reported. The approach is discussed in more detail in [SeX03]. The interpretation of the profiles follows the outline given in Section 3.

The incidents are interpreted by the architects and acted upon accordingly. If the violations are premeditated due to e.g. performance issues, no action needs to be taken. If the incidents are real architectural violations, the corresponding changes must be made, both to the architecture model in case of an existing forward or round-trip engineering process, and to the implementation model. These changes, in turn, are reflected to the reverse engineering process. If the incidents are found non-relevant, the profiles and/or validation configuration must be modified. The tool, *artDECO* [SeX03][PeS04]

[1] Rigi: http://www.rigi.csc.uvic.ca/

together with a suitable validation configuration performs the validation of the architectural views against the architectural profiles in practice.

Model processing operations in the analysis process
In addition to the profile-based architecture validation process, a set of model processing operations [SKS03] can are used for analyzing and reasoning about the architectural views. The model processing operations (e.g. transformations, set operations, abstractions) can be used for generating new transient views to the system under observation. Ideally, they result in increased program comprehension, based on which additional decisions on the development of the system can be done. The model processing operations are built on top xUMLi.

A particularly interesting category of model processing operations in the context of this work are set operations [Sel03][Ven04]. The set operations can be used, among other things, for comparing different model versions, and for performing model slicing based on a given criterion. If a later architecture model version is compared against an earlier one, the differences (i.e., added, removed, and migrated parts) can be focused on when estimating the impact of the changes. The same applies even if the second model is a forward engineering model or a reference architecture description, given that a correspondence relationship between them can be established. The deviations can then be seen as potential architecture violations, analogously to the artDECO incidents. The role of the set operations and model comparison is further increased, should there exists a forward engineering model to be simultaneously maintained. A similar technique has been applied on comparing the reverse engineering capabilities of UML-based CASE-tools [Kea02].

Other categories of model processing operations can be used as well. Projection operations can be used for generating dedicated slices of the architectural views. Transformation operations can be used for migrating information: for example, if the reverse engineering process is able to produce complimentary execution traces in the form of UML sequence diagrams, these diagrams, when transformed into structural representations [SKS03], can be used as a slicing criterion for slicing the original model using the set operations. Model processing operations and their usage has been addressed e.g. in [SKS03][PeS01].

6 THE MAINTENANCE PROCESS IN PRACTICE: MODEL ANALYSIS IN THE ISA CASE STUDY

The target system for the maintenance process, hereafter referred to as ISA, consists of a large-scale product platform architecture and real-life product-line products built on top of this platform. The domain is embedded software for mobile products. Roughly, the target architecture model has around 150 subsystems, 1000 components, and 15000 dependencies.

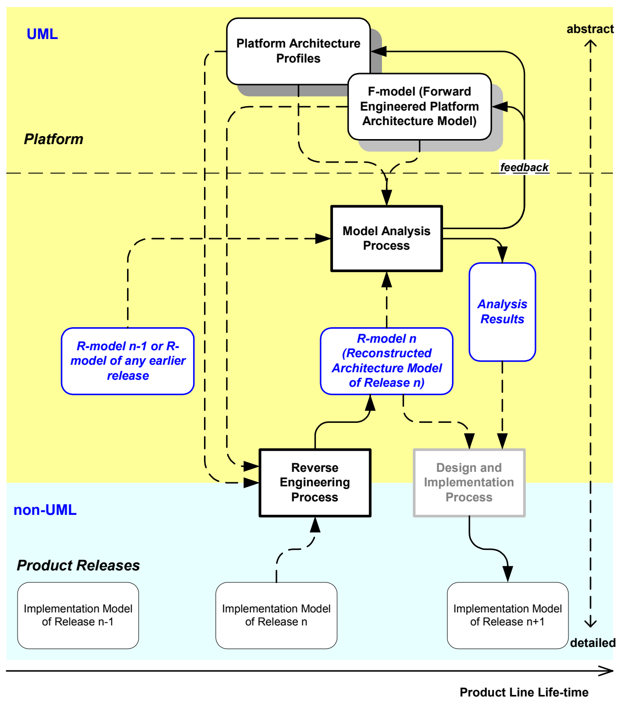
UML
Platform Architecture Profiles
F-model (Forward Engineered Platform Architecture Model)
abstract
Platform
feedback
Model Analysis Process
Analysis Results
R-model n-1 or R-model of any earlier release
R-model n (Reconstructed Architecture Model of Release n)
Reverse Engineering Process
Design and Implementation Process
non-UML
Product Releases
Implementation Model of Release n-1
Implementation Model of Release n
Implementation Model of Release n+1
detailed
Product Line Life-time

Figure 10. Maintenance process for the ISA platform

In the ISA case study, a forward engineered model (F-model) and profile platform architectural profiles are used to guide the construction of reverse engineered model (R-model) for further analysis as described earlier. In later maintenance activities, new R-models are constructed and analyzed against the profiles and previous versions of the R-models. Figure 10 shows this maintenance process, which shows an instance of the architecture maintenance process depicted in Figure 1 for the ISA case study.

Profile-based architecture validation
The architectural profiles for the target models have been described in Section 3. Table 1 lists the architectural concerns, together with their UML interpretations, that the artDECO tool has been used for validating.

Table 1. UML interpretations of architectural concerns

Architectural concern	*UML interpretation*
Validate that only the architectural concepts allowed by the domain are being used	Check that all classes, packages, and dependencies have stereotypes defined in the architectural profiles

Validate that only allowed relationships between the architectural concepts are being used	Check that all used dependencies have been defined in the architectural profiles
Validate that the interfaces are correctly realized	Check that the interfaces have been properly defined in the architectural profiles and that there exists proper realizers for them
Validate that the number of relationships between architectural concepts are correct	Check that the number of associations between concepts conforms to the multiplicity definitions in the architectural profiles
Validate that the physical composition hierarchy of the architecture conforms to the domain-specific conventions (i.e., Top-Level packages – Subpackages – Components)	Check that the package containment hierarchy and owned classes follow the parent – child relationships defined in the packaging profiles
Validate that the system conforms to a three-layer architecture style	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

The F- model has been synthesized from existing non-UML architectural descriptions (e.g. databases and Excel sheets). While it provides an interesting starting point, we plan to reconstruct the platform architecture model after we gain enough experience from the application of RE-process. The validation process reported around 100 stereotype violations, 1000 invalid interface dependencies, and 100 illegal dependencies between layers. The latter were found to be relevant problems in the F-model, reflecting the fact that the system was not originally designed to follow a layered architecture style, and most of the dependencies were introduced without following the layering principles.

Most of the other violations were related to stereotype mismatches, but the results still revealed over 100 real violations. Due to the significant number of violations for a design model, a further investigation was performed. The F-model turned out not to be strictly a design model. Many component-level details, especially their dependencies, actually stem from the architects' impressions of the implementation, and reflected many ad hoc solutions in the implementation. It became evident that the architects tried to close the gap between the designed and implemented architecture before the RE-process was applied. The effort obviously failed but resulted in an architecture model residing between the two worlds.

The R-model was produced by the RE-process described in Section 4. After analyzing the initial validation results, improving the RE-process and the MA-process, and refactoring the profiles, the reported incidents included about 300 stereotype violations, 5000 illegal dependencies, and 650 layering violations. All stereotype-related incidents resulted from the usage of architectural types known to be invalid *a priori*, i.e., components and elements the RE-process was unable to map to existing architectural concept.

Of the reported dependencies, around 3700 of the incidents are genuine, i.e., not involving an element whose stereotype is reported invalid. Obviously, many of the violations originated from the quick'n'dirty solutions (e.g. during bug fixing) under the time-to-market pressure. But the initial findings from the investigation still revealed at least the following sources of the problems both in the implementation and the MA- and RE-processes:

1. *Evolving and incomplete profiles*. When new stereotypes were introduced, the constraints of the dependencies regarding them could not be completely specified, and some of them were even left open. The dependencies (usually legal ones) that were not covered by the profiles were reported illegal by the artDECO tool.
2. *Components having inappropriate stereotypes.* Some components with a certain stereotype actually played different roles than what had been defined by the stereotype. This discovery shows that on one hand the hints or rules used by the RE-process to recognize the component types should be improved, and on the other hand the design and implementation of those components should be reviewed.
3. *Sharing of code in the same development team.* Many illegal dependencies occurred between the components that were implemented by the same team or several closely cooperating groups working on the same feature or feature set. As an example, a component can share a function from another component with a function call. This can easily create an illegal dependency regarding to the design principles, simply because the developers can be physically located near each other or the components may even originate from the same developer.
4. *Dependencies introduced due to performance and other non-functional requirements.* Some of the dependencies short-cutting across layers involved top layer components directly invoking the functions of the

bottom layer components by-passing the middle layer. The dependencies of this category are special and considered to be allowed in a real-time and embedded system. However, they must be monitored and controlled closely within a very limited scale.

5. *Inappropriate function allocation.* Our investigation showed that several components controlling global data had been placed in the top layer. Most of the components in the lower layers depended on them, and this was the source of most of the layering violations. The investigation showed that two of them had the client stereotype and one had an unmatchable stereotype. Just these three components were able to cause over 1000 illegal dependencies.

These findings gave valuable feedback for improving the architecture design and implementation, maintenance of the architectural profiles, and the MA- and RE-processes.

Applying UML model processing operations

When a new architecture model is generated, it is often useful to compare it to the previous architecture model versions. In the context of this work, the comparison is done with the set operations. The commonalities and differences between the models are detected and presented to the user both as a (set of) UML diagram(s) and as a summary report, focusing attention of the designer to the variations between the two models. The model against which the new model is compared can either be a previous version of the architecture model, a reference architecture model, or a forward engineering model. The results can be further processed using the other available model operations. Some of the possible changes in the architecture models include addition, removal, change, and migration of components and subsystems, and addition, removal, and change of dependencies.

The application of set operations requires mechanisms for detecting the correspondence of model elements belonging to the two input models. With the ISA case study, this is straightforward: the names of subsystems and components have a global mapping and are thus unique. The dependencies are then identified through their end elements and types. As an example, an interesting subsystem was selected and two consecutive versions were compared.

Another particularly interesting operation is the slicing of the model based on given slicing criteria. As an example, the diagram excerpt shown in Figure 11 shows a small part of an ISA subsystem, sliced against a set of traces. The traces have been gathered by performing selected use cases on instrumented software. The traces, presented as UML sequence diagrams, have been transformed into a class diagram, which in turn is used as the slicing criterion. When accompanied with suitable set of abstraction and filtering operations, the architect can perform various interesting model processing tasks on the target model.

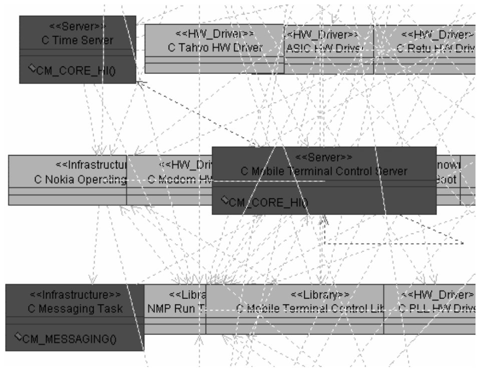

Figure 11. Slicing a subsystem against traces

Examples of interesting operations include slicing of the model based on given slicing criteria (e.g. a subsystem, an OCL expression, sequence diagrams).

7 RELATED WORK

Some techniques and approaches that combine bottom-up and top-down reverse engineering approaches have been developed. In [MuN97] Murphy and Notkin discuss the software Reflextion Model (RM) technique and its application. In this technique, a high-level model is first defined and presented using a special kind of a directed graph. In our approach, we use UML notation for making it easier to map the approach with forward engineering activities. In the RM technique, a model is extracted of the source, a mapping describing how entities in the source and high-level models relate is constructed, and the two models are compared using a set of computation tools. In our approach, the mapping between the source and UML models is guided by the profile descriptions and existing design-level models. As in the RM technique, the mapping is mostly constructed manually. The comparison of the extracted UML models with the existing models (e.g., previous versions) and profiles is carried out using an extensible set of UML model operations built on the top of xUMLi platform. Both our approach and the RM technique are iterative allowing refinements of the models constructed. In addition, we believe that the software analysis techniques are highly influenced by the domain in question and therefore in our approach the reverse engineering and model analysis subprocesses are also refined along the iterations.

In ARES [Gal95], informal domain knowledge, domain standards, and coding guidelines are combined with information derived from source code to support software architecture recovery. The architecture of the subject system is first categorized on a coarse level, similarly to the categorization of difference architectural styles [Sha95]. This categorization is then used to define what concepts are looked for from the code. The information on the different releases of the subject software system is stored in a database. In our approach, the domain knowledge and guidelines are defined using UML profiles. We currently reverse engineer the UML models and compare it with the previous version or with the forward engineered (original) model. Relevant architectural information is stored in a relational database. A Web interface and other visualization techniques are used to provide an easy access to it.

8 DISCUSSION

UML has been widely used for designing and describing software models. However, effective approaches and tool support for UML-based architecture modeling has been lacking [Mea02]. Correspondingly, UML has been used, to some extent, in reverse engineering, but these tools do not really support the construction of architectural models. Instead, they are typically limited to reverse engineering class diagrams [Kea02].

In this paper we proposed an approach to software architecture maintenance and showed how it has been adopted for maintaining a large-scale product platform architecture and real-life product-line products built on top of this platform. The development of the approach has been driven by the practical needs in industry. Experience suggests that maintenance processes are strongly dependent on the domain they are applied to. In our approach, the flexibility needed is gained by using architectural profiles. The profiles provide a powerful means to define the domain-dependent architectural styles and rules that guide the reverse engineering and model analysis subprocesses. We have further aimed at a flexible tool environment, customizable and adoptable for different domains. For instance, the profiles also influence the model validation rules, model manipulation methods, and reverse engineering methods. Therefore, the tools used in this environment are designed and implemented to be customizable and modifiable, and new tools can be conveniently integrated in the environment [Rea04].

ACKNOWLEDGEMENTS

The authors wish to thank Nokia, Jani Airaksinen for the implementation of the artDECO tool kit, Jan Salvador van der Ven for his work on the set operations. The work of Claudio Riva and Jianli Xu are also funded by FAMILIES (Eureka Σ! 2023 Programme, ITEA project ip02009).

REFERENCES

[Deur04] A. van Deursen, C. Hofmeister, R. Koschke, L. Moonen and C. Riva. Symphony: View-Driven Software Architecture Reconstruction. In *Proc. of the IEEE/IFIP Working Conference on Software Architecture (WICSA'04)*. IEEE Computer Society, 2004.

[Aea02] J. Airaksinen, K. Koskimies, J. Koskinen, J. Peltonen, P. Selonen, M. Siikarla, and T. Systä, "xUMLi: Towards a Tool-Independent UML Processing Platform", In K. Østerbye (Ed.), *The Proc. of NWPER* , 2002, pp. 1-15.

[Gall95] H. Gall, R. Klösch, and R. Mittermeir, Object-Oriented Re-Architecting, In *Proc. of ESEC 1995*, LNCS, Springer-Verlag, September 1995, pp. 499-519.

[Kea02] R. Kollmann, P. Selonen, E.Stroulia, T. Systä, and A. Zündorf, A Study on the Current State of the Art in Tool-Supporter UML-Based Static Reverse Engineering, In *Proc. of WCRE 2002*, pp. 22-33.

[Mea02] N. Medvidovic, D.S. Rosenblum, D.F. Redmiles and J.E. Robbins, Modeling Software Architecture in the Unified Modeling Language, *ACM Transactions on Software Engineering and Methodology*, Vol 11, No 1, January 2002.

[MuN97] G.C. Murphy and D. Notkin, Reengineering with Reflextion Models: A Case Study, *IEEE Software*, 1997.

[PeS04] J. Peltonen, and P. Selonen, "An Approach and a Platform for Building UML Model Processing Tools", In *Proc. of ICSE 2004 Workshop WoDiSEE2004*, 2004, pp. 51-57.

[Riv00] Riva C., Reverse Architecting: an Industrial Experience Report, In *Proc. of WCRE2000*, 2000, pp. 42-51.

[Rea04] C. Riva, P. Selonen, T. Systä, A.-P. Tuovinen, J. Xu , and Y. Yang, Establishing a Software Architecting Environment, In *Proc. of WICSA 2004*, 2004.

[RXM01] Riva C., Xu J. and Maccari A., Architecting and Reverse Architecting in UML, In *Proc. of ICSE 2001 Workshop on Describing Software Architecture with UML*, 2001.

[OMG02] The Object Management Group, Unified Modeling Language Specification (Action Semantics) – UML 1.4 with Action Semantics, Final Adopted Specification, January 2002. On-line at http://www.omg.org/uml.

[PeS01] J. Peltonen, and P. Selonen, "Processing UML Models with Visual Scripts", In *Proc. of HCC'01*, 2001, pp. 264-271.

[SKS03] P. Selonen, K. Koskimies, and M. Sakkinen, Transformations Between UML Diagrams, *Journal of Database Management*, 14(3), Idea Group Publishing, 2003, pp. 37-55.

[SeX03] P. Selonen P and J. Xu, Validating UML Models Against Architectural Profiles. In *Proc. of ESEC 2003*, 2003, pp. 58-67.

[Sha95] M. Shaw, Comparing architectural design styles, *IEEE Software*, November 1995, pp. 27-41.

[Ven04] J. van der Ven, An Implementation of Set Operations on UML Diagrams, M.Sc. thesis, University of Groningen, Instituut voor Wiskunde en Informatica, 2004.