A Framework for Software Architecture Refactoring using Model Transformations and Semantic Annotations*

Igor Ivkovic and Kostas Kontogiannis
Dept. of Electrical and Computer Engineering
University of Waterloo
Waterloo, ON N2L3G1 Canada
{iivkovic, kostas}@swen.uwaterloo.ca

Abstract

Software-intensive systems evolve continuously under the pressure of new and changing requirements, generally leading to an increase in overall system complexity. In this respect, to improve quality and decrease complexity, software artifacts need to be restructured and refactored throughout their lifecycle. Since software architecture artifacts represent the highest level of implementation abstraction, and constitute the first step in mapping requirements to design, architecture refactorings can be considered as the first step in the quest of maintaining system quality during evolution. In this paper, we introduce an approach for refactoring software architecture artifacts using model transformations and quality improvement semantic annotations. First, the conceptual architecture view is represented as a UML 2.0 profile with corresponding stereotypes. Second, instantiated architecture models are annotated using elements of the refactoring context, including soft-goals, metrics, and constraints. Finally, the actions that are most suitable for the given refactoring context are applied after being selected from a set of possible refactorings. The approach is applied to a simple example, demonstrating refactoring transformations for improved maintainability, performance, and security.

Keywords: *software evolution, software architecture refactoring, model transformations, UML profiles, quality-driven re-engineering*

1. Introduction

Software evolves in reaction to various environmental stimuli including variations in business and technical requirements, and changes in market demands and conditions. Within the context of model-driven software evolution, changes can be applied to software models at different levels of abstraction and detail, such as architecture design or source code models. As a result to these changes, the complexity and brittleness of software grows, and inconsistency between related models and their elements may arise. To decrease complexity and ensure integrity of the different artifact models, it is necessary to create a controlled environment for software evolution where model transformation and model refactoring mutations can be performed systematically.

Refactoring in a traditional sense refers to restructuring the source code of the system. In model-driven architecture (MDA) and encompassing model-driven development (MDD), refactoring can be performed on any MOF-compliant artifact model, and is generally viewed as a process of improving the internal structure of a software system without disrupting its external behavior [10]. We refer to a refactoring mutation as a neutral software evolution cycle that is based on specific quality goal that does not affect external system behavior. Based on this view, refactoring also refers to interpreting quality goals, defined as nonfunctional requirements, in terms of structural changes of artifacts models under specific preconditions (*e.g.*, existence of a specific design pattern) and postconditions (*e.g.*, must not change specific interaction scenario).

In forward engineering, software architecture artifacts represent the first step in mapping requirements models to design elements. Similarly, from the reverse-engineering perspective, software architecture is among the highest level source code abstraction models that can be extracted. Consequently, in a top-down approach to software refactoring, architectural refactoring can be considered as the first step in changing system implementation to match specific quality-improvement goals. Changes to architectural models must, however, be performed carefully and systematically, as they may impact the elements of more specific

[1] This work is funded in part by the IBM Canada Ltd. Laboratory, Center for Advanced Studies (CAS) in Toronto.

design and implementation models. Nevertheless, if performed correctly and accurately, changes at the architectural level may lead to more significant, and positive improvements in the structure of a software system.

In this paper, we introduce an approach for refactoring of software architecture models using semantic annotations and UML 2.0 profiles. We use profiles instead of domain models or metamodels to create a unified method for refactoring of different models and views that allows preservation of metamodel properties, such as types and relations, but that also allows expression of refactoring transformations in a generic way. As a first step in realizing this goal, software architecture views are represented using UML 2.0 profiles, where specific view elements are denoted by restrictive stereotypes. These stereotypes extend or modify the abstract syntax of language elements, and define the usage and semantics for the newly added elements [2] (*e.g.*, <<component>> and <<connector>>). Similarly, the soft-goals, metrics, and constraints associated with a particular refactoring quality goal are denoted by the *refactoring context* UML profile, and made part of the <<semanticHead>> stereotype to represent a set of related soft-goals and metrics. A semanticHead instance is linked to a refactoring transformation to indicate its effect on soft-goals and applicable metrics. Next, using an approach to quality-driven refactoring as introduced by Mylopoulos *et al*. [6], a specific refactoring goal is represented as a soft-goal interdependence graph (SIG) of soft-goals and related quality metrics. Using the refactoring context, the elements of SIG are mapped to architectural refactoring transformations. Finally, based on a query of semantic annotations, the most suitable refactoring transformations are selected and applied.

1.1. Motivation

By introducing the proposed framework, we aim to address the following:

1. Standardize and formalize quality-driven refactoring using UML. We formalize existing refactoring methods through the refactoring context and model transformations.

2. Use UML profiles to represent different architecture modelling views in a standardized format. As a proof of concept, we have created a profile for the conceptual architecture view.

3. Show a method for encoding refactoring transformations and use them as part of the framework to refactor an existing system. We provide a comprehensive example on which we illustrate encoding, selection, and application of transformations.

This paper is organized as follows: Section 2 outlines related research in the areas of architectural repair and refactoring and model transformations. Section 3 discusses the representation of software architecture and architectural refactoring using UML 2.0 profiles. Section 4 introduces our approach to software architecture refactoring using soft-goals and defined UML profiles and applies it to a detailed example, with Section 5 that provides our conclusions and directions for future research.

2. Related Research

Mens [17] has conducted a detailed survey of software refactoring. As part of the survey, the general approach to refactoring was identified to include: (1) identify the location for refactoring, directly on the source code or on one of the more abstract artifacts; (2) determine which refactorings should be applied for the given location, for instance, refactoring *bad smells* such as code duplication to improve maintainability [10]; (3) guarantee that the refactoring changes that are considered preserve external behavior, for example, using preconditions and postconditions defined on the same set of input and output values [19]; (4) apply selected refactorings, (5) assess the effect of the refactoring on the quality (*e.g.*, complexity, or performance) or the process (*e.g.*, cost, or effort) using metrics, controlled experiments, and statistics; and (6) maintain the consistency between the refactored elements and other software artifacts using different consistency management techniques such as mSynTra [13] or Fujaba [9]. In this paper, we adopt the abstract model of refactoring, and we create the refactoring context to represent the key elements for model-driven software refactoring. For example, the refactoring location is identified as the source rule for the refactoring model transformation while specific behavioral constraints are encoded as pre- and post-conditions.

The structure of individual refactoring transformations is based on a classification of model transformations by Czarnecki and Helsen [8]. That is, each transformation rule is composed of two parts: a source rule and a target rule, expressed as patterns of strings or graph elements, or logic constraints. Transformations can also be endogenous, if they transform models in the same language, or exogenous, if they transform models between different languages. We define refactoring transformations as the mapping of the source and target graph patterns represented in UML.

Other approaches to refactoring include quality-driven reengineering approaches [22], where basic transformations are used to compose more complex ones based on their relation to specific soft-goals. We extend these approaches by creating a formalism for representing and selecting generic refactorings, and relating them in a structured manner to specific quality goals. We also note UML-based refactor-

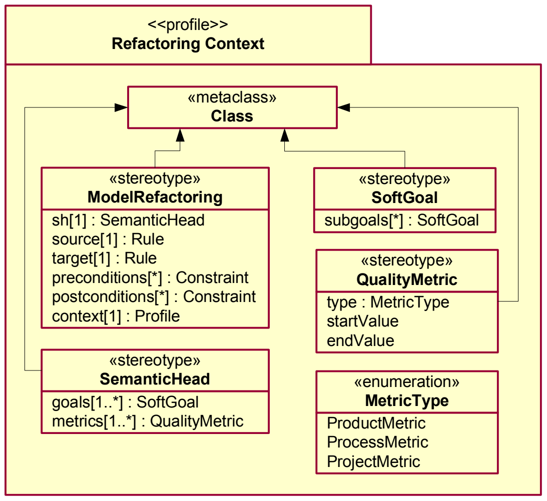


Figure 1. Refactoring Context Profile

ing approach [21], and previously defined UML profiles for requirements [11], and product lines [24].

3. Defining the Refactoring Context for Software Architecture using UML Profiles

In this section, we first introduce and describe elements of a general refactoring context for software evolution. We then use a metamodel for describing software architecture views as introduced by Hofmeister *et al.* [12] to create UML profiles for different architectural views, namely the conceptual architecture view profile. Finally, we apply the created profiles to define the refactoring context for software architecture artifacts.

3.1. Generic Refactoring Context

To define a generic refactoring context for software evolution, we make use of a refactoring process based on multiple soft-goals [23], which represent nonfunctional requirements and factors that may affect them.

The steps in this process are:

- Step 1. Starting with a set of refactoring goals, represent the reasoning model for refactoring as a soft-goal interdependence graph (SIG), which represents dependencies between soft-goals. Use the SIG to quantify acceptance or rejection attributes of each soft-goal as a $[0, 1]$ metric.

- Step 2. Measure quantifiable properties of soft-goals to decide the ordering of relevance in the soft-goal hierarchy, and select the one that should be applied first.

- Step 3. Pick the most suitable refactoring transformation based on their contribution to the chosen soft-goal.

- Step 4. Apply the selected refactoring. If there are still soft-goals that were not addressed, return to Step 2 and repeat the evaluation, ordering, and selection. Else, terminate the process.

Based on the above process and related research [11, 17], we propose the following elements as stereotypes to be part of the semantic representation of a refactoring context. Figure 1 illustrates the resulting Refactoring Context as a UML profile.

- <<stereotype>> **ModelRefactoring**

  Description
  Represents a semantically annotated refactoring transformation.

  Attributes
  sh[1] : SemanticHead — semantic annotation
  source[1] : Rule — a source pattern
  target[1] : Rule — a target pattern
  preconditions[*] : Constraint — the conditions that must be satisfied for a refactoring to apply
  postconditions[*] : Constraint — the conditions that must be satisfied for a refactoring to terminate successfully
  context[1] : Profile — a corresponding refactoring perspective

  Semantics
  Each ModelRefactoring represents one semantically annotated refactoring transformation. Selection of refactorings is based on the values stored in the SemanticHead attributes.

- <<stereotype>> **SemanticHead**

  Description
  Represents semantic annotation for ModelRefactoring.

  Attributes
  goals[1..*] : SoftGoal
  metrics[1..*] : QualityMetric

  Semantics
  Refactoring transformations are associated with corresponding SemanticHead instances. Selection of refactorings is based on the values stored in the SemanticHead attributes.

- <<stereotype>> **SoftGoal**

  Description
  Represents quality goals for refactoring. One or many SoftGoal instances are a part of the SemanticHead for the refactoring context.

Attributes
subgoals[*] : SoftGoals — subordinate SoftGoals

Semantics
A refactoring context is represented through a set of soft-goals derived from SIG. To satisfy the refactoring goals, each of the soft-goals has to be iteratively mapped to a corresponding ModelRefactoring, which is then applied.

- <<stereotype>> **QualityMetric**

Description
Represents quality metrics for refactoring. One or many QualityMetric instances are a part of the SemanticHead for the refactoring context.

Attributes
type : MetricType
startValue — computed metric value for the source pattern
endValue — computed metric value for the target pattern

Semantics
Each of the refactoring context soft-goals is measured using an appropriate quality metric. Depending on the differences between starting and ending values, which represent metric values for the source and target graph patterns respectively, the chosen soft-goals are matched to individual refactorings.

- <<enumeration>> **MetricType**

Description
Represents different quality metric categories, namely ProductMetric, ProcessMetric, and ProjectMetric, as defined in [15].

Semantics
Each of the QualityMetric is typed using an instance of MetricType.

3.2. Conceptual Architecture View Profile

To define a UML profile for software architecture, we treat each software architecture view type individually. That is, we consider the following architectural views as defined in [12]:

- Conceptual architecture view — contains components, connectors, ports, roles, and protocols, where ports, roles, and protocols may be abstracted as parts of specific interfaces [7].

- Module architecture view —- encompasses models, subsystems, layers, and interface protocol.

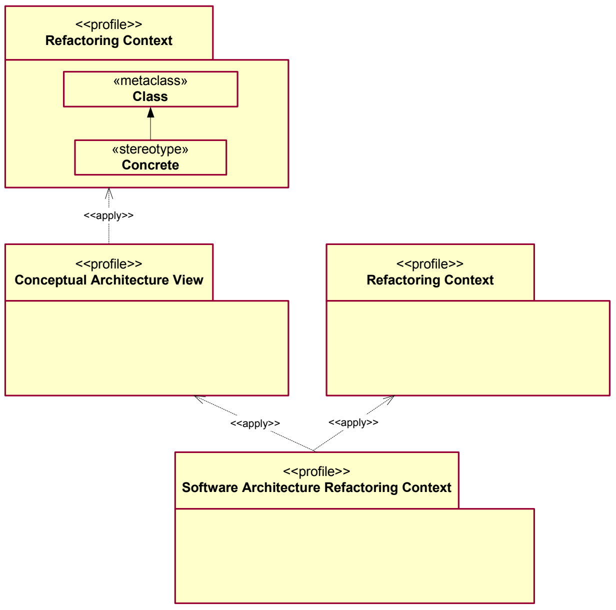


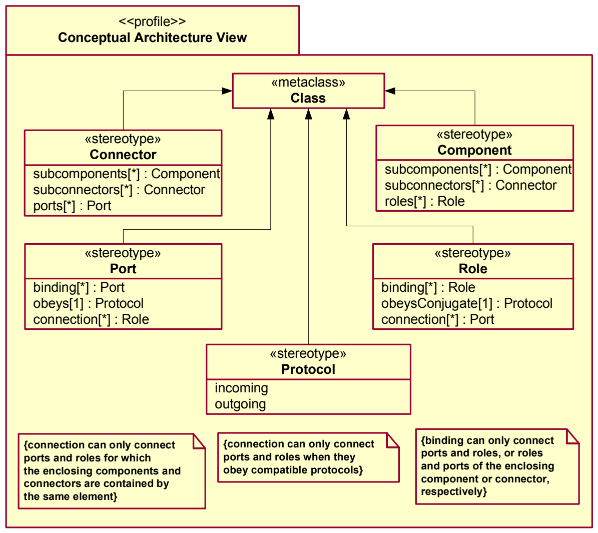
Figure 3. Refactoring Context for the Conceptual Architecture View

- Execution architecture view — with run-time images, communication paths, and communication protocols.

- Code architecture view — including source files, intermediate files, executable files, and directories.

In this paper, we focus on the conceptual architecture view, and define a UML profile for it using the previously published metamodel and semantics [12]. Based on the UML 2.0 specifications, association relations between stereotypes in a profile are not allowed. However, there is no constraint on showing associations as stereotype attributes, so we have used attributes to represent associations between stereotypes to create the profile shown in Figure 2. The full semantic description of the profile is not provided, as it is analog to the description for the original Hofmeister *et al*. metamodel. Instead, we provide an extension to this metamodel to account for the differences between the "as-designed architecture", which refers to the conceptual architectural model created during the architectural design stage, and the "as-implemented architecture", which refers to the architectural model that was recovered from the implementation artifacts [3]. The differences between these two views are indicative of possible architectural drift that may arise due to evolution of software artifacts, and are detrimental to maintainability.

From the described UML profiles, now we can define the refactoring context for software architecture. Figure



<<profile>>
Conceptual Architecture View
«metaclass»
Class
«stereotype»
Connector
subcomponents[*] : Component
subconnectors[*] : Connector
ports[*] : Port
«stereotype»
Component
subcomponents[*] : Component
subconnectors[*] : Connector
roles[*] : Role
«stereotype»
Port
binding[*] : Port
obeys[1] : Protocol
connection[*] : Role
«stereotype»
Role
binding[*] : Role
obeysConjugate[1] : Protocol
connection[*] : Port
«stereotype»
Protocol
incoming
outgoing
{connection can only connect ports and roles for which the enclosing components and connectors are contained by the same element}
{connection can only connect ports and roles when they obey compatible protocols}
{binding can only connect ports and roles, or roles and ports of the enclosing component or connector, respectively}

Figure 2. Conceptual Architecture View Profile

3 illustrates the refactoring context for conceptual architectural view as a combination of the generic refactoring context and conceptual architectural view profiles. According to this view, individual refactorings are stereotyped and annotated based on the refactoring context, that is, each transformation is represented as a combination of the <<source>> and <<target>> stereotyped rules, along with an instance of <<semanticHead>> stereotype. Elements that are used in the source and target patterns are stereotyped according the elements of the Conceptual Architecture View profile (*e.g.*, <<component>> and <<connector>>). The rule elements may be additionally stereotyped as <<concrete>> to annotate those elements that are identified (or confirmed) through software architecture recovery. Refactoring contexts for other views are defined in the same manner, as a combination of the generic refactoring context and a specific architectural view profile. The resulting profile is used to represent specific model refactorings, as shown in Section 4.

4. A Framework for Software Architecture Refactoring

In this section, we make use of the refactoring context to create a framework for software architecture refactoring. First, we introduce the steps in the refactoring process, from identifying refactoring context elements to validating the applied transformations and terminating. We then demonstrate the approach on a comprehensive example of architectural refactoring.

4.1. Software Architecture Refactoring Process

Using the elements of the refactoring context and the multiple soft-goals refactoring from [23], we identify the following steps in our approach.

Step 1. Identify Soft-Goal Hierarchy
Identify the semantic annotation elements including a hierarchy of soft-goals. Denote the hierarchy as a soft-goal interdependence graph (SIG).

Step 2. Identify Architectural Refactoring
Derive a set of candidate architectural refactorings for the particular architectural view, with soft-goal hierarchy as guidance. For instance, apply suitable Fowler's refactorings [10] to software models, and identify those that lead to expected improvements in specific soft-goals, such as increased performance. Represent each refactoring as a model transformation rule, with the source and target patterns, preconditions and postconditions constraints, and specific architectural view to which they are applied.

Step 3. Associate Refactorings with Semantic Heads
Annotate derived refactorings with compatible metrics, and compute their values. Use differences in source and target metric values to establish applicable soft-goals. For instance, a decrease in a coupling metric would indicate a positive effect on modularity.

Step 4. Select Primary Soft-Goal and Apply Refactorings

Choose the next primary soft-goal based on the

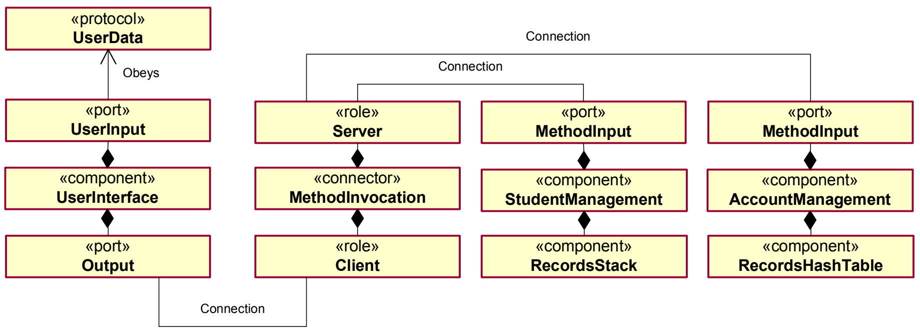


Figure 4. Architectural Refactoring Example: Original Configuration

soft-goal hierarchy. Select and apply the refactorings with the greatest contribution towards the selected goal. For possible collisions in selection, select the refactoring with the highest positive effect on the largest number of soft-goals. Before selecting the transformation, verify that the preconditions hold, and before committing the change, validate the transformation using postconditions.

Step 5. Process Next Soft-Goal or Terminate
If there are still soft-goals that were not addressed, go to Step 4. Otherwise, report success and terminate.

4.2. Applying the Refactoring Framework

As a demonstrative example, let us consider the following case of architectural refactoring of a software system responsible for management of student accounts in a university or college organization (see Figure 4). The system contains two main components: StudentManagement and AccountManagement, reflecting two main use cases for the system: manage student information, and manage account information. Both components store their data into individual, internal databases using internal data components: RecordsStack for storing student data, and RecordsHashTable for storing account data. Under the increasing loads on the database to retain information about old students for up to ten years, and due to increasing needs of the university to accommodate more students every year, the performance of the system has degraded significantly.

To improve the performance of the system, it is necessary to replace the internal data components with either an optimized external data structure component, or a modern database-management system. Moreover, to improve the maintainability of the system, it is desirable to decrease component coupling, and apply a model-view-controller (MVC) architecture style. Finally, access to both databases is not controlled, and there is no access authentication. To improve security, it is desirable to institute role access policies to alleviate security concerns. To address these goals, we apply our software architecture refactoring framework.

4.2.1 Step 1. Identify Soft-Goal Hierarchy

From the considered soft-goals, we identify the following sample soft-goal hierarchy as shown in Figure 5.

As the first soft-goal, we have identified maintainability — aptitude of the system to undergo maintenance and evolution [1] — based on the combined views of [22]. The top-level goal of high maintainability is interpreted as high changeability in the context of this example. We have further identified two subgoals: high modularity, and low change complexity, which we have mapped through subgoals to corresponding metrics. As a general resource for metric identification, we have used a metric categorization from [15]. Specifically for modularity, we have used Coupling Between Object (CBO), as the number of classes to which a given class is coupled, and Lack of Cohesion on Methods (LCOM), as the number of disjoint sets of local methods, metrics that we introduced by Chidamber and Kemerer (CK) as part of their CK metrics suite [5].

For the second goal, we have derived performance — the ability of a system to allocate different computation resources to respond to service requests while satisfying timing requirements [7] — using the soft-goal graph from [22]. The top-level goal of high performance is interpreted as high time and high space performance, and mapped through subgoals to corresponding metrics. Given that the focus of performance-driven refactoring in this example is the database system component, we have hence chosen corresponding database performance metrics: number of queries processed per time unit for low response time, and the amount of data read per time unit for high throughput [18].

For the third goal, we have identified security — protection of system data from disclosure, modification, or destruction — based on the corresponding security hierarchy

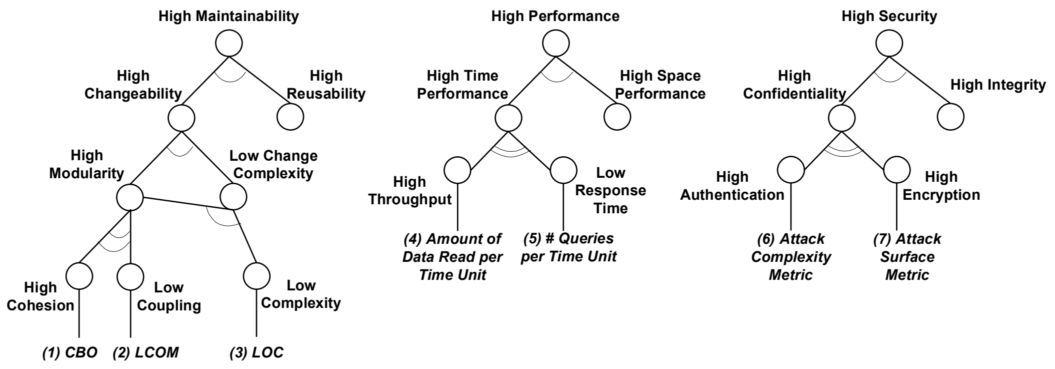


Figure 5. Derived Soft-Goal Graph

[1]. The top-level goal of high security is interpreted as high confidentiality and high integrity, and mapped through subgoals to corresponding metrics. For high authentication, we have chosen attack complexity metric [4], which estimates the complexity of effort invested by attacker to cause a security breach, including different attack mechanisms. For high encryption, we have selected attack surface metric [16], which indicates the number of system resources of selected type available for attack from the outside.

4.2.2 Step 2. Identify Architectural Refactoring

Based on the analysis of the given system, we have identified desired refactoring actions, and compared them to Fowler's classification of source-code refactorings [10]. As a result of the comparison, we have identified the following desired transformations for the system:

- (Rule1) Separate Database from Component (see Figure 6), where a target database component is coupled and internally contained by its parent. To improve modularity, this complex transformation (1) removes the component from the parent, (2) instantiates it as a new optimized component with an appropriate connector, and (3) reestablishes the connection between the target and the parent. One of the important preconditions is for the source rule to be correctly matched against a pattern in the source model to which it is applied. Based on the stereotypes, a pattern of two components, a parent that contains a child component, is identified. However, it is important to apply the pattern to only those child components that provide database functionality.

  To solve this mapping problem, we have used an approach to matching model elements using semantic annotations [14]. The matching rules can also be expressed as OCL constraints on features defined

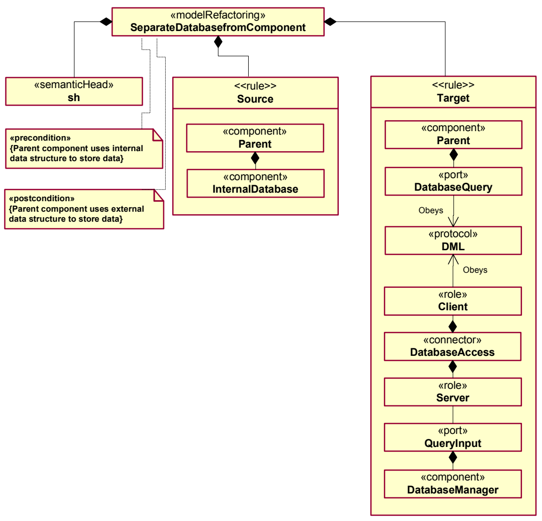


Figure 6. Separate Database from Component Rule

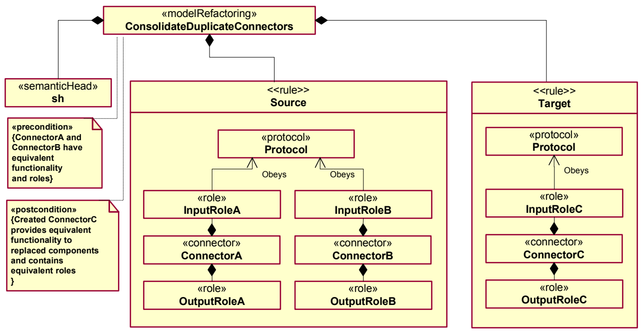


Figure 7. Consolidate Duplicate Connectors Rule

through a feature map. The source pattern would then be matched with a location in the applied model only if the child component of the connector implements the same feature (*i.e.*, database access). A similar approach that uses feature models and information retrieval techniques was proposed in [20]. However, the topic of feature mapping and element matching is out of the scope of this paper, and will be addressed in more detail in future research.

- (Rule2) Consolidate Duplicate Components / Connectors (see Figure 7), where duplicate components or connectors are unified into one. To decrease change complexity by decreasing the size of the system, this complex transformation (1) identifies functionality-equal components or connectors, (2) removes the redundant instances, and (3) reconnects the connectors or components to the now unified component or connector, respectively.
- (Role3) Implement Role Access Policies (see Figure 9), where access to a resource is controlled through a specific connector that implements a role access policy. To improve security by increasing the complexity of attack through authentication, this complex refactoring (1) identifies a connector that provides access to one or more components without role authentication, (2) creates different roles within the connector with separate roles and role access policies for each resource, and (3) reconnects the connected components to newly created roles.

4.2.3 Step 3. Associate Refactorings with Semantic Heads

For each of the defined transformation rules, we now instantiate corresponding semantic heads. For each rule, we measure values for each of the metrics identified in Step 1. The results are interpreted as positive, neutral, or negative effects that each refactoring has on a specific soft-goal.

- For Rule1, we observe (1) a decrease in the CBO metric for the database component from one to zero; (2) decrease from two to one in the LCOM metric for the parent of the database component, but no change in the relative LCOM value since a new component was created; (3) undeterminable or no significant effect on the LOC metric; (4 and 5) increase in the amount of data read per time unit and number of queries per time unit, by exact values that depend on the optimization of the new component; (6) increase in the attack complexity metric, since an attacker now cannot access the database by only compromising the parent component; and (7) undeterminable or no effect on the attack surface metric, since no additional protection was specified. The results of the observations are significantly positive effects on (++) maintainability and (++) performance, and some positive effects on (+) security.
- For Rule2, (1 and 2) no effect on the CBO and LCOM metric since the connecting protocols did not change (for CBO) and since only a connector was removed (for LCOM); (3) decrease in the LCOM metric since a redundant component was removed; (4 and 5) no effect on the performance metrics since the individual data flows did not changes; (6 and 7) no effect on the security metrics since no additional protection was specified. The results of the observations are a positive effect on (+) maintainability, and no determinable effect on performance and security.
- For Rule3, (1) no effect on the CBO metric since the connecting protocols did not change; (2) increase in the LCOM metric from one to two with addition of another set of local methods for handling the second access policy; (3) increase in the LOC metric with code for handling additional roles and role access policies; (4 and 5) decrease in the performance metrics since additional authentication processing was added; (6) increase in the attack complexity metric by adding authentication; and (7) decrease in the attack surface metric, since resources protected by authentication are not directly exposed. The results of the observations are negative (or very negative, depending on specific authentication scenarios) effects on maintainability (-/–) and performance (-/–), and very positive effects on security (++).

In the given annotation analysis, we have not focused on specific metric values but instead on trends in their change to determine the relations between model refactorings and soft-goals. This approach simulates practical circumstances, where due to time constraints it may not be possible to accurately define and apply each metric. Moreover, the analysis of each metric may need to include not just the trends in the refactoring transformation but also effects on the system to which the transformation is applied. Hence, it would be desirable if specific metrics could be associated as part of UML profiles for specific refactoring types (*e.g.*, performance metrics for performance-improving transformations), and automatically computed when specific transformations are selected and applied. Extending profiles with specific metric sets is part of our future research.

4.2.4 Step 4. (Iteration 1) Select Primary Soft-Goal and Apply Refactorings

In this first iteration of applying selected refactorings, we use *modifiability* as our primary soft-goal. From Step 3, re-

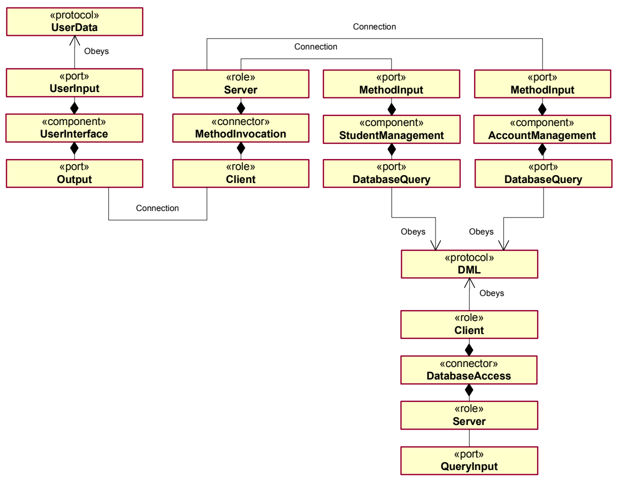


Figure 8. Architectural Refactoring Example: Intermediate Configuration

lated transformations are Rule1 and Rule2, where Rule1 has more significant effect on modifiability (++). Therefore, we apply (1) Separate Database from Component transformation once, and (2) Consolidate Duplicate Connectors transformation twice (once for duplicate component, and once for duplicate connector) to achieve improvement in modifiability. The result of applying the three transformations to the system, without showing intermediate steps for space consideration, is shown in Figure 8.

4.2.5 Step 4. (Iteration 2) Select Primary Soft-Goal and Apply Refactorings

In the second iteration, we select *performance* as our primary soft-goal. However, we determine that performance was already significantly improved through the effects of Rule1. Additional constraints (postconditions) on performance metrics could be used to quantifiable verify that this soft-goal is satisfied.

4.2.6 Step 4. (Iteration 3) Select Primary Soft-Goal and Apply Refactorings

In the third iteration, we select *security* as our primary soft-goal. We determine that security was already addressed partially through the effects of Rule1. From Step 3, we now identify Rule3 as having significant effect on security (++), and we apply it to the configuration from Figure 8. The resulting structure after applying this rule is shown in Figure 10.

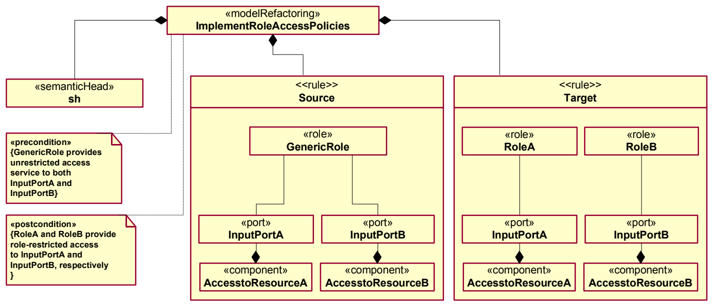

«modelRefactoring» ImplementRoleAccessPolicies
«semanticHead» sh
«precondition» {GenericRole provides unrestricted access service to both InputPortA and InputPortB}
«postcondition» {RoleA and RoleB provide role-restricted access to InputPortA and InputPortB, respectively }
<<rule>> Source
«role» GenericRole
«port» InputPortA
«port» InputPortB
«component» AccesstoResourceA
«component» AccesstoResourceB
<<rule>> Target
«role» RoleA
«role» RoleB
«port» InputPortA
«port» InputPortB
«component» AccesstoResourceA
«component» AccesstoResourceB

Figure 9. Implement Role Access Policies Rule

4.2.7 Step 5. Process Next Soft-Goal or Terminateg

Since all of the primary soft-goals were address, we declare success with the final configuration shown in Figure 10, and terminate.

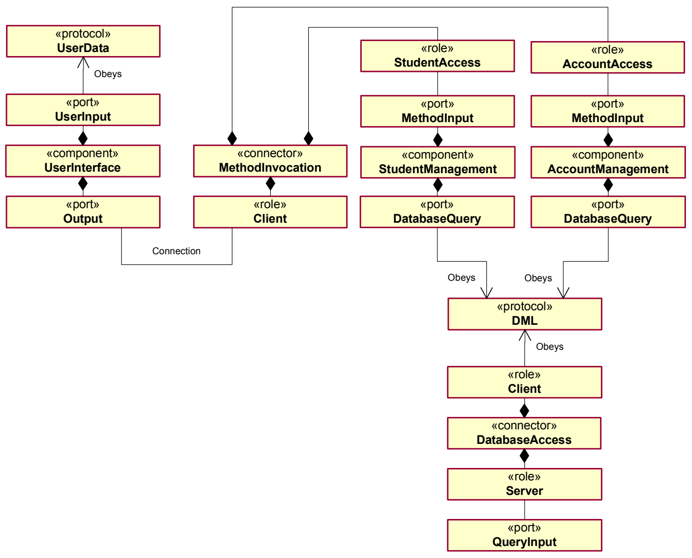


Figure 10. Architectural Refactoring Example: Refactored Configuration

5. Conclusions and Future Research

In this paper, we have presented an approach to refactoring of software architecture artifacts using model transformations and semantic annotations related to system quality improvements. We have defined a generic refactoring context using UML 2.0 profiles, including

<<semanticHead>> stereotype for denoting semantic annotations. We have also instantiated a profile for the conceptual architecture view, as an example of a unified method for representing different architectural view notations. Finally, we have used the refactoring context and the conceptual architecture view profile to define specific refactoring transformations.

In future research, we plan to extend the framework by applying it to more complex case studies. The extensions will address additional semantic elements for representing individual model annotations to allow more precise transformation rule expression, and quantification through metrics of the quality improvements presented in SIGs. We also intend to apply the approach to refactoring of other software artifacts, including software requirements and concrete design, and define specific UML profiles for different artifact views.

6. Acknowledgements

This work is supported and performed in collaboration with the IBM Canada Ltd. Laboratory, Center for Advanced Studies (CAS) in Toronto.

References

[1] M. Barbacci, M. Klein, T. Longstaff, and C. Weinstock. Quality attributes. Technical Report CMU/SEI-95-TR-021, Software Engineering Institute (SEI), Carnegie Mellon University, Pittsburgh, PA, December 1995.

[2] S. Berner, M. Glinz, and S. Joos. A classification of stereotypes for object-oriented modeling languages. In *Proceedings of the Second International Conference on The Unified Modeling Language (UML 1999)*, Fort Collins, CO, Oct 1999.

[3] I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a case study: Its extracted software architecture. In *Proceedings of the International Conference on Software Engineering (ICSE 1999)*, Los Angeles, CA, May 1999.

[4] S. Brocklehurst, B. Littlewood, T. Olovsson, and E. Johsson. On measurement of operational security. In *Proceedings of the Annual Conference on Computer Assurance*, 1994.

[5] S. R. Chidamber and C. F. Kemerer. A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering (TSE)*, 20, 1994.

[6] L. K. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Publishing, 2000.

[7] P. Clements, R. Kazman, and M. Klein. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley, Indianapolis, IN, 2002.

[8] K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, Anaheim, CA, Oct 2003.

[9] T. Fischer, J. Niere, L. Torunski, and A. Zndorf. Story diagrams: A new graph rewrite language based on the unified modeling language. In *Proceedings of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT 1998)*, Paderborn, Germany, Nov 1998.

[10] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Upper Saddle River, NJ, 2000.

[11] W. Heaven and A. Finkelstein. Uml profile to support requirements engineering with kaos. *Software, IEE Proceedings*, 151(1), Feb 2004.

[12] C. Hofmeister, R. Nord, and D. Soni. *Applied Software Architecture*. Addison-Wesley, Reading, MA, 2000.

[13] I. Ivkovic and K. Kontogiannis. Tracing evolution changes through model synchronization. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM 2004)*, Chicago, IL, Sep 2004.

[14] I. Ivkovic and K. Kontogiannis. Towards automatic establishment of model dependencies. *Accepted to the International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, Sep 2005.

[15] S. H. Kan. *Metrics and Models in Software Quality Engineering*. Addison Wesley, Boston, MA, 2003.

[16] P. Manadhata and J. M. Wing. An attack surface metric. Technical Report CMU-CS-05-155, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Jul 2005.

[17] T. Mens. A survey of software refactoring. *IEEE Transactions on Software Engineering (TSE)*, 30(2), Feb 2004.

[18] B. B. Mortensen. Beyond response time: Detailed database performance analysis. Master's thesis, University of Copenhagen, 2004.

[19] W. Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[20] I. Pashov, M. Riebisch, and I. Philippow. Supporting architectural restructuring by analyzing feature models. In *Proceedings of the Eighth European Conference on Software Maintenance and Reengineering (CSMR04)*, Tampere, Finland, Mar 2004.

[21] C. Riva, P. Selonen, T. Syst, and J. Xu. Uml-based reverse engineering and model analysis approaches for software architecture maintenance. In *Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM 2004)*, Chicago, IL, Sep 2004.

[22] L. Tahvildari and K. Kontogiannis. A software transformation framework for quality-driven object-oriented reengineering. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2002)*, Oct 2002.

[23] Y. Yu, J. Mylopoulos, E. Yu, J. C. Leite, L. L. Liu, and E. D'Hollander. Software architecture styles as graph grammars. In *Proceedings of the The First International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE 2003)*, Victoria, Canada, Nov 2003.

[24] T. Ziadi, L. Helouet, and J. Jezequel. Towards a uml profile for software product lines. In *Proceedings of the 5th International Workshop on Software Product-Family Engineering (PFE 2003)*, Siena, Italy, Nov 2003.