OPTIMIZING NON-FUNCTIONAL PROPERTIES OF A SERVICE COMPOSITION USING A DECLARATIVE MODEL-TO-MODEL TRANSFORMATION

Andreas Petter, Stephan Borgert, Erwin Aitenbichler, Alexander Behring and Max Mühlhäuser

Abstract. Developing applications comprising service composition is a complex task. Service composition requires the knowledge of various process languages (e.g. WS-BPEL, XPDL, or WSFL) or the knowledge of languages like WS-CDL which focus more on messaging aspects. To choose the right language for the problem at hand requires a lot of research as different aspects of various languages need to be considered. Therefore, to lower the skill barrier for developers it is important to describe the problem on an abstract level and not to focus on implementation details. This can be done using declarative programming which fosters to describe only the result of the problem (which is what the developer wants) rather than the description of the implementation. We therefore use purely declarative model-to-model transformations written in a universal model transformation language which is capable of handling even non-function properties using optimization and mathematical programming. This makes it easier to understand and describe service composition and non-fuctional properties for the developer. [1]

Keywords:*declarative programming, business processes, model-to-model transformation*

2000 *Mathematics Subject Classification*: 68R10, 90C35

[1]The project was partly funded by means of the German Federal Ministry of Economy and Technology under the promotional reference 01MQ07012 and partly by the SoKNOS project which is funded by the German Federal Ministry of Education and Research. The authors take the responsibility for the contents.

1. Introduction

Developing applications that perform service composition is complex. To reduce the complexity of application development, model driven development has been investigated in the past. Even though model transformation has also been used for service composition (e.g. [3]) approaches do not use optimization and a declarative universal transformation language. However, both are needed to perform rapid prototyping and application development of service compositions with non-functional properties in a declarative way:

Optimization must be performed to calculate an optimal combination of services which lead to a service composition which is either maximal or minimal given a set of non-functional properties.

If a proprietary model transformation language is used (e.g. [3]) instead of a universal transformation language, developers will need to learn a new language for developing their service composition. Therefore the speedup in the development phase is reduced because of the time which is needed to learn the new language. In contrast, our approach is based on on a language based on QVT Relations which is a standard model-to-model transformation language. Developers do not need to learn a new language, but only minimal changes.

A declarative language is needed to focus the way a developer understands his service composition problem at hand on the outcome rather on the way how to implement it. In contrast, an imperative language will probably foster thinking about the way how to implement it. We believe that this helps developers to focus more on an understanding of the problem itself and prevents slipping into implementation details. Especially, this will be important if trade-offs need to be taken into consideration. In Section 3 we present a target function to select services according to a set of non-functional properties. We strived that this target function can just be copied over into the implementation of the model-to-model transformation with only very minor changes (only the names of the variables). Developers can then focus on the definition of the target function and can almost forget how to implement it.

1.1 Definitions and Contributions

In this paper we use classical definitions of service composition, processes and services. However, for clarification we repeat the definitions in an informal way. An *application model* is a model which describes temporal relationships between a set of processes. A *process* is a procedure or a function that can

be called either local or remote and has at least one distinct and well-defined functional interpretation which is known by all entities performing service composition.

Usually a single service composition is not sufficient and a developer wants service compositions to be computed for various application models.

Definition A *composition program* is a program that computes service compositions for a set of application models.

A composition program may also take into account *non-functional properties*, which describe aspects of services which have no direct impact on their interpretation. Three examples, which are considered in this paper are:

- the costs to execute a service if it is a premium service which needs be payed for
- the runtime to execute the service call
- and a measure how reliable the service is, e.g. the uptime of the hosting machine

This paper focuses on the creation of composition programs using a universal and declarative model-to-model transformation language in combination with optimization.

In this paper we make the following contributions:

- Presenting a purely declarative way to develop composition programs,
- using a universal model-to-model transformation language with constraint solving and optimization to perform service composition and
- by doing that, we present a fast way to develop service composition programs.

1.2 Document Structure

This document is structured as follows: Section 2 gives an overview over our scenario. Our theory to combine composition programs with optimization and how we combine different non-functional properties is presented in Section 3 while our approach to meta-modelling for this scenario is presented in Section 4. Then the transformation language is presented in Section 5 as well as an implementation in Section 6. The paper closes with an overview of the state of the art 7 and our conclusions and future work in Section 8.

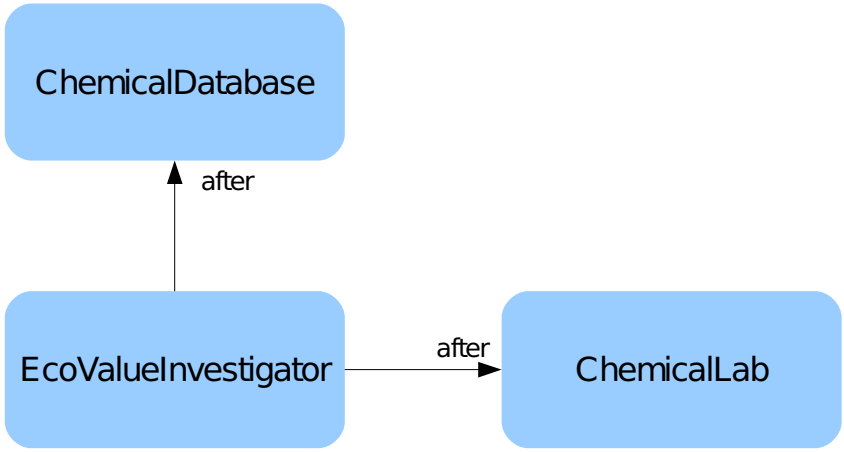

Figure 1: Abstract Model (Application Model)

2. Scenario

In this work, we chose an "Eco Calculator" as a running example (see Figure 1).

In our scenario engineers may discover one or multiple such services ("Eco Calculators) on the Internet and send construction details of, e.g., a car seat to this service. The service then calculates the ecological impact of production, recycling, etc. of the product and automatically issues compliance certificates accordingly. The behavior of the Eco calculator service is described by a formal business process. Because the process involves knowledge of various disciplines, we assume that the provider of the Eco Calculator service is willing to outscore certain parts of the process.

Hence, we used three different types of services for demonstration purposes: the Eco Calculator service itself, a chemical database service, and a chemical laboratory service. All services are described by formal business processes.

The whole process model is now formed by a service composition that integrates all three different types of services. For each service type, we have modeled five different variants, which all differ a bit in their behavior to come

as close to practical situations. In practice, there are also services which show an equivalent behavior on an abstract level, but differ in small details. For example, every Eco Calculator service can receive construction details and issue certificates, but it can be different in its administrative processes such as delivery or payment.

It is important to note that this is just a running example and our approach is not dependant on any of the concepts presented in this section. Also the result of the service is not important at development time, so the "ecological impact" is not important for the users of the model-to-model transformation, but only for the users of the whole composition.

3. Optimization and Non-Functional Properties

In a composition program services need to be assigned to processes. For each process there must be at least one service that implements it, c.f. if there isn't any service a part of the application model will not be executable. Therefore, there has to be a mapping from processes to services. This mapping is performed in the composition program, therefore the mapping either needs to be done in the model-to-model transformation itself or in a separate model that describes the mapping. For simplicity we present the first version, while the second is just a simple extension of the concept (which can be implemented by either generating the transformation or by inserting another level of abstraction within the transformation itself). In our case the mapping will be implemented in relations which map a process to a set of possible target services.

Each member of the set of possible target services has a number of non-functional properties, which can easily be extended by changing the models (c.f. Section 4). The problem we want to solve is, what set of services is an optimal set according to the properties presented in the introduction (execution costs, runtime and reliability). For simplicity we assume that all properties are given using integer values and that the lower the value the better (so for reliability this is anti-proportional to the uptime of the service). Then we can easily sum up all properties of the same type (e.g. all execution costs called "rentingCosts" in our scenario) of all services S currently used in our composition:

$$R_i = \sum_{s \in S} rentingCosts \tag{1}$$

If this is done for all compositions $i = 1..n$ we can compare all R_i and e.g. calculate the minimum R_i:

$$\min(\{R_i|\forall i \in \{1..n\}\}) \tag{2}$$

Things get more complicated if several non-functional properties come into play. Then we need to compare the results of all properties, i.e. they must be comparable. As this is a difficult question and is subject to continous testing and experience we chose to just multiply the results of each property with a coefficient representing a weight chosen by the developers. Then we added the results into one big sum (let T_i be the sum of all runtimes and S_i the sum of all relabilities and a, b, and c factors):

$$\min(\{a * R_i + b * T_i + c * S_i|\forall i \in \{1..n\}\})) \tag{3}$$

Choosing a, b and c is very difficult and requires much experience in the domain of service composition.

We will see that the target function can easily be implemented in a model-to-model transformation using Solverational by copying most of the formula.

4. Meta-Models of Composition Programs and Services

Seen from a model-driven engineering perspective a composition program is a model-transformation either model-to-model or model-to-code. While both types of transformations are possible, we focus on the model-to-model transformation process. In that case the concrete models produced by the transformation can be changed more easily and can be better understood than pure code (which would be the result of a model-to-code transformation). The concrete models can then be transformed into code by a relatively simple model-to-code transformation.

In our case a service composition is a model-to-model transformation from an application model to a service model. The application model, we call it "abstract model" uses the business processes that should be performed by the application as model elements. It is the source model for the model-to-model transformation. In our case it is comprised of an EcoCalculator investigator process, a chemical lab process and a chemical database process. It is shown in figure 1 presented in Section 2.

To define the model we defined an abstract meta-model (see Figure 2) which contains all the processes which can be used to model the behavior of the application, called "abstract model". Then, the abstract model is an instance of the abstract meta-model. To model the behavior the model elements

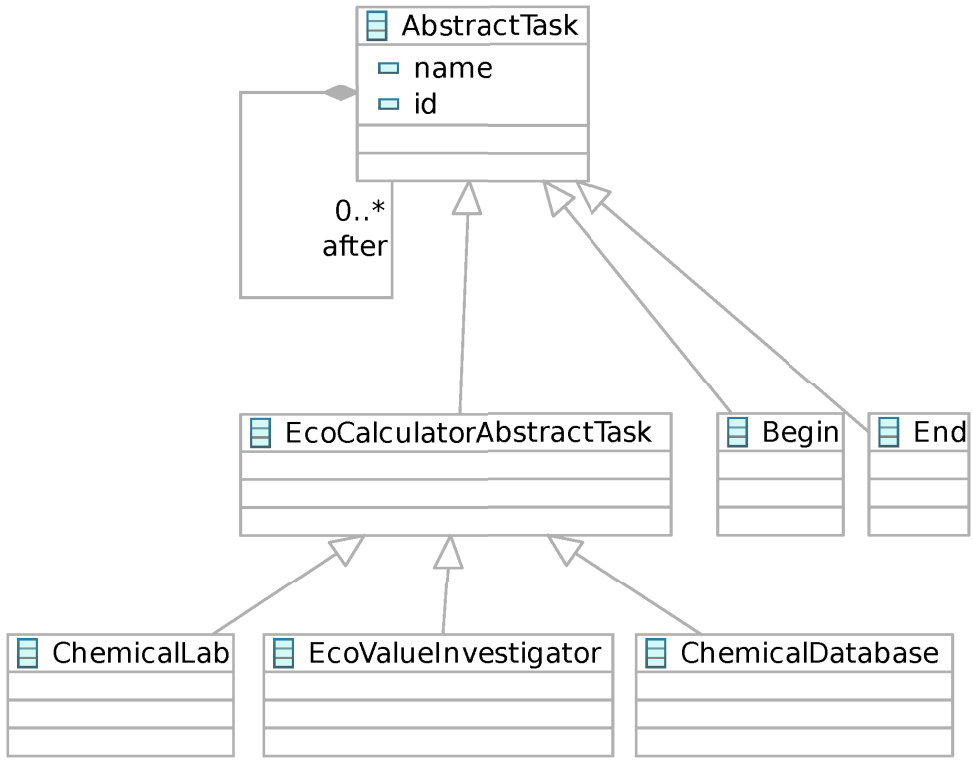

Figure 2: Abstract Meta-Model

of the abstract model (which represent processes) we instantiate sub-classes of the "AbstractTask and use the "after" association to define a temporal relationship between other processes. Basically the "after" association models that a message is being sent over the network and the process which is the target of the association is triggered. All processes sub-class the domain specific "EcoCalculatorAbstractTask" which is perfectly modelled after our scenario. Any new domains will introduce new meta-model elements sub-classing "AbstractTask" in a way which is specific to their domain.

The abstract model is transformed into a concrete one by replacing the processes from the abstract model with concrete service implementations. This model can be defined using the concrete meta-model shown in Figure 3. In this meta-model we defined all possible services for the EcoClaculator scenario.

These are implemented as subclasses of the "EcoCalculatorTask".

All possible services must subclass "Task" which defines the "after" and "before" associations which have the same meanings as in the abstract model (the before association is an opposite association for after). Each service then has three cost values (these are just examples; any other cost values can be implemented by adding an additional cost properties to "Task") which can be handled according to Section 3.

5. Model-to-Model Transformation with Solverational

We assumed that the Solverational model-to-model transformation language can be used for prototyping service compositions in a purely declarative way. Therefore, we developed a transformation performing service composition and comprising non-functional properties using Solverational as a proof of concept.

5.1 Solverational

Solverational [7] is based on QVT Relations [5], which is a standard defined by the OMG.

QVT Relations is a declarative model-to-model transformation language. It uses a relational programming approach to declarative programming. Thereby, a QVT Relations transformation uses any number of relations to relate source and target meta-model elements. If a relation is applied on a source model (in our case the "abstract model"), the transformation selects a set of model elements to be transformed into a set of target model elements. This set may be constrained by so called "PropertyTemplateItems" which allow for the definition of expressions which must be satisfied by property values of model elements. Relations can again be related to each other by defining dependencies (e.g. when a relation holds for a model then another relation must hold, too).

Developers of transformations do not explicitly state the creation of new model elements. Only the transformation engine is in charge of creating new model elements. Thereby the approach is known to be very declarative and abstract [2], which is good for focusing on the definition of the problem at hand and not its implementation.

Implementations will usually be written in JAVA and are based on EMF (Eclipse Modelling Framework) of the Eclipse JAVA IDE and therefore use

models noted in EMOF ([4]) as input as well as output (c.f. they use the Ecore meta-meta-model which is part of EMF).

Solverational extends this approach by allowing inequalities for "PropertyTemplateItems". Thereby the whole transformation can be viewed as a means to define constraint solving problems [7]. Additionally, Solverational allows for the definition of target functions to perform mathematical optimization resulting in "constraint optimization problems". The transformation engine in fact is a compiler which compiles Solverational transformations into ECLiPSe (Eclipse Constraint Logic Programming System [1]; not to be confused with the Eclipse JAVA IDE) code which is then executed using ECLiPSe. Models are represented in ECLiPSe logic terms while being executed and are re-transformed into EMOF (Ecore) models to provide output to the Eclipse JAVA IDE.

5.2 Solverational: Alternative Domains

In Solverational relations can have several target domains. These target domains represent the sets of model elements which need to be created in the transformation process. They must be marked as being "alternative" domains, and the transformation engine needs to choose an alternative domain during the transformation process.

This will usually be done by providing the target function. If several alternatives exist, the target function will "guide" the transformation engine in selecting the optimal domain and therefore force it to create the optimal model element for the case at hand. An example is given in the next section.

6. Implementation

We implemented the transformation using the Solverational model-to-model transformation language.

Figure 4 presents some details of the transformation. The alternative domains known from the section 5.2 are prominent. Here the relation which transforms the "ChemicalLab" into the concrete services uses one alternative domain for each concrete service. The transformation engine chooses the "best" alternative by means of mathematical optimization.

As presented in Section 3 the transformation uses an optimization function which was just copied over from the formulas presented in Section 3. For demonstration purposes we chose the values $a = 5$, $b = 2$ and $c = 3$.

6.1 Performance of the Model-to-Model Transformation

As Solverational provides an universal transformation language applications developed in Solverational are usually not as fast as special solutions which use e.g. optimizations in assembly language or other low level languages. There are still comparatively few services available on the internet, which can be used for our service composition. The approach is probabaly fast enough under these circumstances.

Although our implementation therefore is not the fastest implementation it is a very fast and simple way to implement an implementation at all. I.e. the developer has not as much work as with implementing the transformation in a low level language.

Our scenario is comprised of 15 services; resulting in 125 combinations, which can be computed (enumerated) very fast. The runtime is fast enough to even run the transformation at runtime, so the services and the target meta-model can easily be generated at runtime to support dynamic service composition. However, if more properties are being added and more services become available runtime will potentially increase exponentially. Then, if it has been applied to real world applications, developers would need to re-implement the transformation in an imperative language and would think about further optimizations.

7. State of the Art

The combination of model driven engineering, declarative techniques and service composition programs have only been researched recently.

The most promising approach was performed by Montanari and Rossi who used a graph rewriting system in combination with constraint solving to describe systems and services [3]. However, they did not use a universal model-to-model transformation language and did not perform optimization for non-functional properties.

Zhu et al. present an approach using UML to describe semantic aspects of web services in OWL-S [10]. They use an MDA [6] based approach to generate Promela description which allow for verification of the composition using the SPIN tool. However their approach is very complex and the transformation is not done declaratively. Our approach could learn from the complex verification process that is applied in this work.

Quintero and Vincente present conceptional models which can be used in an MDA approach to model the mappings of processes to services [8]. In further works these models could be used as an abstraction layer to factor out the mapping from the transformations. However, as Quintero and Vincente present conceptional models used in a specific tool called OOWS this approach is not solely based on a declarative approach to model transformation, but requires a specific tool.

Using model weaving White et al. present an approach which is most similar to our own [9]. They develop the JAVA Pet store application using model weaving and implement the weaving process using a constraint solving problem. However, they use model weaving and do not use a declarative model-to-model transformation which is based on a standard model-to-model transformation language like QVT Relations.

To sum up, all approaches investigated seem to strive for a model driven engineering approach to developing applications using services, but none is using a universal model-to-model transformation language together with optimization. As Solverational is the only declarative model-to-model transformation language which at present is able to perform optimization this was not surprisingly new to us.

8. Conclusions and Future Work

We presented a model-to-model transformation which is able to perform service composition while taking into account non-functional properties. The transformation is purely declarative, which enables developers to focus more on the problem itself and the understanding of the problem domain. The relatively complex task of developing a composition program is reduced to stating facts about it. This allows for faster development, as was the case with the implementation of our scenario which was developed in 45 minutes, only.

In the future we will investigate more the coverage of the transformation language in terms of service composition and composition programs.

References

[1] Krzysztof R. Apt and Mark Wallace. *Constraint Logic Programming using Eclipse*. Cambridge University Press, New York, NY, USA, 2007.

[2] Fréderic Jouault and Ivan Kurtev. On the architectural alignment of atl and qvt. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1188–1195, New York, NY, USA, 2006. ACM Press.

[3] Ugo Montanari and Francesca Rossi. Graph rewriting, constraint solving and tiles for coordinating distributed systems. applied categorical structures. *Applied Categorical Structures*, 7:7–333, 1999.

[4] OMG. Meta object facility 2.0 core final adopted specification. OMG, October 2003.

[5] OMG. Meta object facility (mof) 2.0 query/view/transformation specification. OMG, July 2007. ptc/07-07-07.

[6] J. Mukerji OMG, J. Miller. Mda guide version 1.0.1. OMG, June 2003. document number: omg/2003-06-01.

[7] Andreas Petter, Alexander Behring, and Max Mühlhäuser. Solving constraints in model transformations. In Richard F. Paige, editor, *Theory and Practice of Model Transformations*, volume 5563/2009 of *Lecture Notes in Computer Science*, pages 132–147. Springer Berlin / Heidelberg, June 2009.

[8] Ricardo Quintero and Vicente Pelechano. Conceptual modeling of service composition using aggregation and specialization relationships. In *ACM-SE 44: Proceedings of the 44th annual Southeast regional conference*, pages 452–457, New York, NY, USA, 2006. ACM.

[9] Jules White, Jeff Gray, and Douglas C. Schmidt. Constraint-based model weaving. *Transactions on Aspect-Oriented Software Development*, open:open, 2009. to appear.

[10] Zhengdong Zhu, Yanping Chen, Ronggui Lan, and Zengzhi Li. Study of mda based semantic web service composition. *Information Technology Journal*, 8:903–909, 2009.

Andreas Petter, Technische Universität Darmstadt, Hochschulstr. 10, 64289 Darmstadt, Germany,
email: *andreaspetter@tk.informatik.tu-darmstadt.de*

Stephan Borgert, Technische Universität Darmstadt, Hochschulstr. 10, 64289 Darmstadt, Germany,
email: *stephan@tk.informatik.tu-darmstadt.de*

Erwin Aitenbichler, Technische Universität Darmstadt, Hochschulstr. 10, 64289 Darmstadt, Germany,
email: *erwin@tk.informatik.tu-darmstadt.de*

Alexander Behring, Technische Universität Darmstadt, Hochschulstr. 10, 64289 Darmstadt, Germany,
email: *behring@tk.informatik.tu-darmstadt.de*

Max Mühlhäuser, Technische Universität Darmstadt, Hochschulstr. 10, 64289 Darmstadt, Germany,
email: *max@tk.informatik.tu-darmstadt.de*

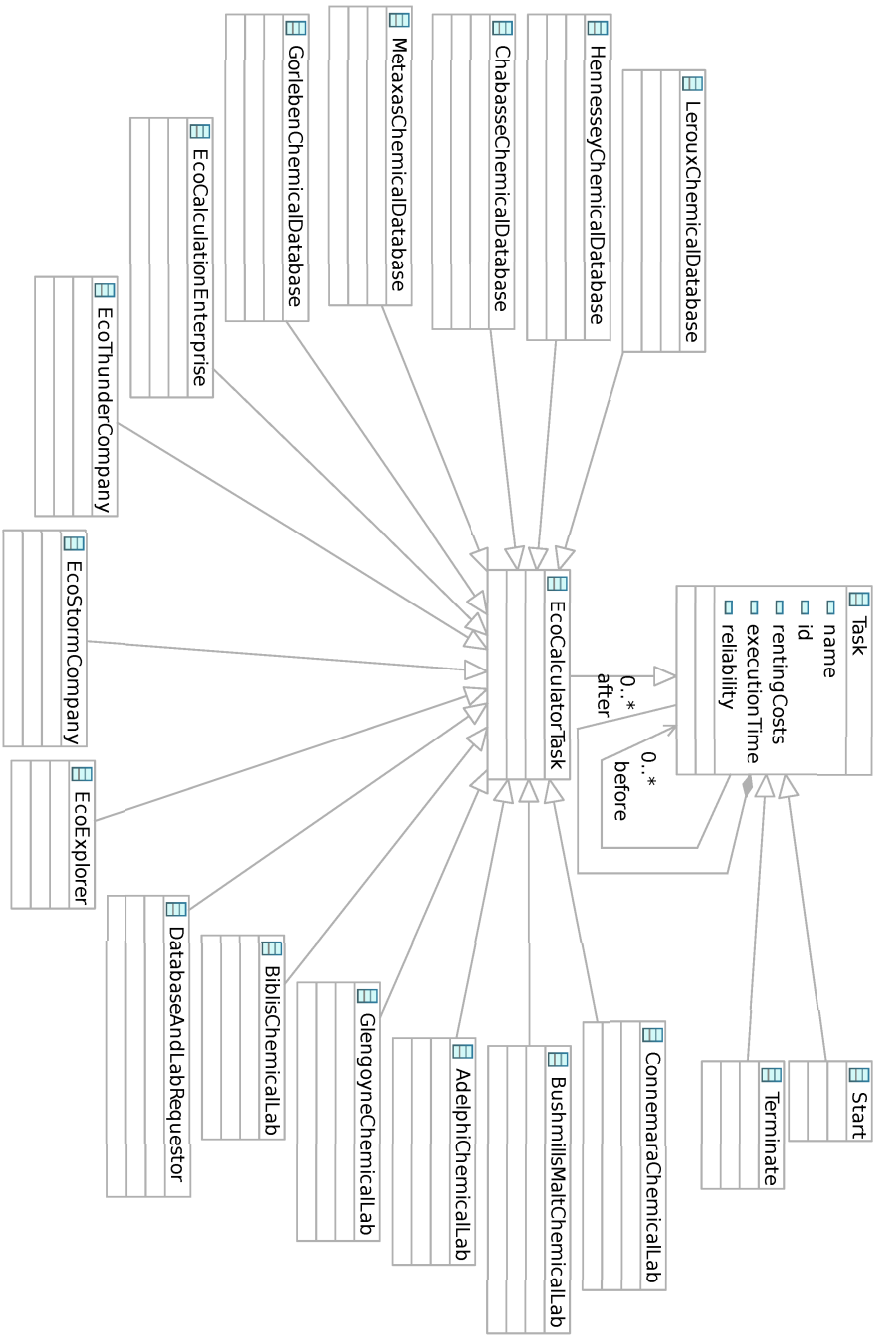

Figure 3: Concrete Meta-Model

```
. . .
minimize 5*sum(concreteTaskMM::Task.allInstances()->rentingCosts) +
         2*sum(concreteTaskMM::Task.allInstances()->executionTime) +
         3*sum(concreteTaskMM::Task.allInstances()->reliability);
. . .
top relation ChemicalLabToLabService {

    domain s i:abstractTaskMM::ChemicalLab { };

    alternative domain t o:concreteTaskMM::AdelphiChemicalLab {
        rentingCosts=1500,
        executionTime=200,
        reliability=300
    };
    alternative domain t o:concreteTaskMM::BiblisChemicalLab {
        rentingCosts=150,
        executionTime=2000,
        reliability=300
    };
    alternative domain t o:concreteTaskMM::BushmillsMaltChemicalLab {
        rentingCosts=500,
        executionTime=350,
        reliability=300
    };
    . . .
}
. . .
```

Figure 4: Transformation details