BRAIN. Broad Research in Artificial Intelligence and Neuroscience,
ISSN 2067-3957, Volume 1, October 2010,
Special Issue on Advances in Applied Sciences,
Eds Barna Iantovics, Marius Măruşteri, Rodica-M. Ion, Roumen Kountchev

UNDERSTANDING SERVICE COMPOSITION WITH NON-FUNCTIONAL PROPERTIES USING DECLARATIVE MODEL-TO-MODEL TRANSFORMATIONS

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

Abstract. Developing applications comprising service composition is a complex task. Therefore, to lower the skill barrier for developers, it is important to describe the problem at hand on an abstract level and not to focus on implementation details. This can be done using declarative programming which allows 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 functional properties using optimization and mathematical programming. This makes it easier to understand and describe service composition and non-functional properties for the developer.

Keywords: *declarativity, business processes, model-to-model transformation*

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

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., [4]) 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 leads to a service composition which is either maximal or minimal to some set of non-functional properties.

If a proprietary model transformation language is used (e.g., [4]) 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 a language, called "Solverational". This language is 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. We believe that this helps developers to focus more on an understanding of the problem itself and prevents slipping into implementation details. This is especially important if trade-offs need to be taken into consideration. In Section 3 we present a objective function to select services according to a set of non-functional properties. It was our goal that this objective function could 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 do not have to worry about its implementation.

1.1 Definitions and Contributions

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 1. *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 to 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 declarative 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.

2. Scenario

In this work, we chose an *Eco Calculator* as a running example. Here, a government agency establishes a new "eco label" for cars meeting certain ecological requirements. One of the requirements is a concept for disassembling and recycling and the restricted use of certain environmentally harmful materials. The service provides a compliance check and cost simulation.

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 outsource 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 limited to this scenario in any way. Also, the result of the service is not important at development time. E.g., concepts like "ecological impact" are entirely domain-specific and completely independent from the generic model-to-model transformation.

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, otherwise 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. 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 specified as integer values and that lower values are better (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 calculate the minimum R_i:

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

This can easily be implemented using Solverational by just copying most of the formula into the transformation code.

4. Meta-Models of Composition Programs and Services

From the perspective of model-driven engineering, 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 only. In that case the concrete models produced by the transformation can be changed more easily and can be better understood than pure code. 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, called *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.

To define the model we first defined the abstract meta-model which contains all the processes which can be used in the application model. These are used to model the behaviour of the application (see Figure 1). The abstract

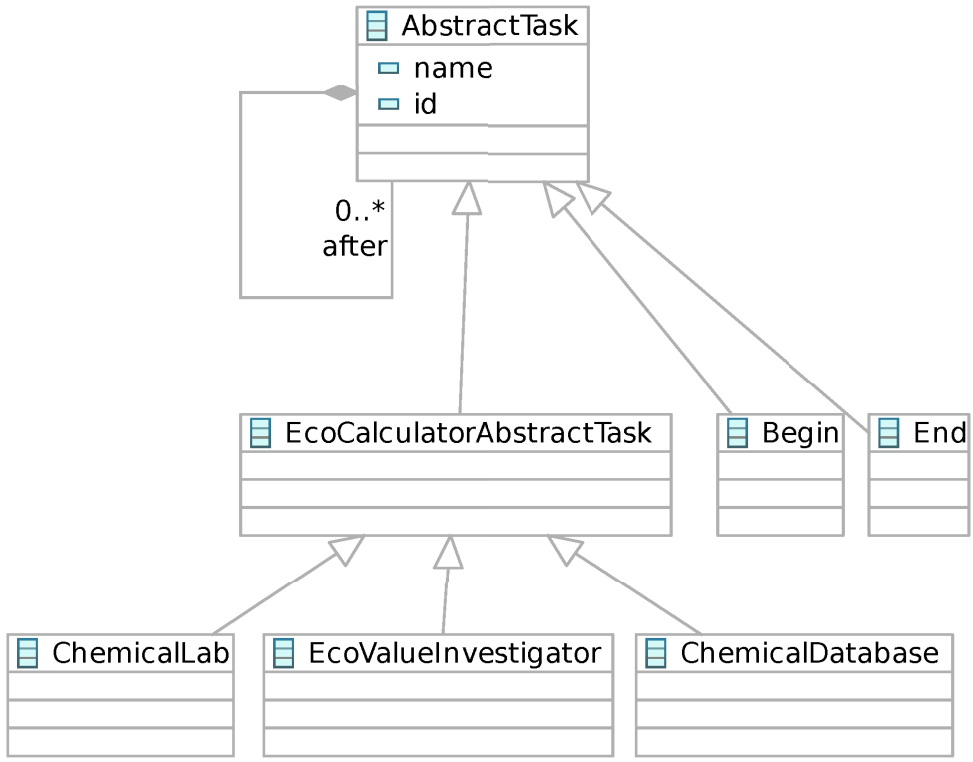

Figure 1: Abstract Meta-Model

model is an instance of the abstract meta-model. To model the behaviour, the model elements of the abstract model (which represent processes), we instantiate sub-classes of 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 being 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 specific to their domain.

The abstract model is transformed into a concrete one, which replaces the processes from the abstract model with concrete service implementations. This model can be defined using the concrete meta-model shown in Figure 2. In

this meta-model we defined all possible services for the EcoCalculator 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 a additional cost properties to Task).

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. We developed a transformation performing service composition comprising non-functional properties using Solverational as a proof of concept.

Solverational [6] is based on QVT Relations [5], which is an OMG standard. 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 QVT Relations is known to be very declarative and abstract [3], which is good for focusing on the definition of the problem at hand and not its implementation.

Solverational extends this approach by allowing inequalities for PropertyTemplateItems. Thereby the whole transformation can be viewed as a means to define constraint solving problems [6]. Relational transformation languages comprising constraint solving on attributes are called *constraint relational transformation languages* [6].

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 objective function (see Section 5.1). If several alternatives exist, an objective 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 Section 6.

5.1 Optimization in Solverational

Optimization can be applied to model-to-model transformations. This allows to select "optimal" target models. Solverational is the first declarative model-to-model-transformation language which supports the declaration of target functions making optimization an integral part of transformations. Its grammar can be found in [7]. Here we derive some general results obtained by including this feature into any model-to-model-transformation language. As noted above this eases especially the understanding and implementation of service composition programs.

Starting from the definition of *constraint satisfaction problems* over models which has been studied in [2] we define constraint optimization problems for model-to-model transformations.

A constraint satisfaction problem over models is a set of constraints C and a set of models. The variables, which are used in the constraints either represent attribute values, associations or model elements (or sets of one of these types). A model will be a valid solution for C, if it satisfies all constraints in C.

Definition 2. *A constraint satisfaction problem over models can either be*

- over-constraint*, i.e. there is no solution to the problem (also called "infeasible")*
- under-constraint*, i.e. there are many solutions to the problem*
- well-defined*, i.e. there is exactly one solution to the problem*

This notion can be extended towards a notion of constraint satisfaction problems in connection with model-to-model transformations: in [6] we defined *constraint relational transformations*. We generalize our notion:

Definition 3. *Let $\mathfrak{M}_s$ be a set of source meta-models, $\mathfrak{M}_t$ be a set of target meta-models, C a set of constraints on the target model and $T(\mathfrak{M}_s, \mathfrak{M}_t, C)$ be a transformation definition which contains a set of constraints C over the set of model elements, attributes or associations. Then $T(\mathfrak{M}_s, \mathfrak{M}_t, C)$ is a* constraint model-to-model transformation problem.

When T is applied to a set of concrete source models conforming to $\mathfrak{M}_t$ each result of a constraint model-to-model transformation problem is a set of constraint satisfaction problem over models, because after the transformation process the constraints out of C apply to the target models M_t. Therefore, we can apply Definition 2: every result of any constraint model-to-model transformation problem is also either over-constraint, under-constraint or well-defined.

Based on this notion we define a notion for optimizing constraint relational transformations.

Optimization problems are comprised of a domain D and an objective function \$, which is to be minimized or maximized, depending on the problem at hand. If $\|D\| > 0$ ($\|X\|$ counts the number of elements in X), then the problem has a solution: a set of domain values for the maximum and minimum values of $\$(D)$.

For *constraint optimization problems* D is given by a set of variables V and constraints C, which restrict the possible values for the domains of the variables. Combining constraint optimization problems and model-to-model transformations requires that D contains models. Then \$ also applies to models.

Definition 4. *A* constraint model-to-model transformation optimization problem $T_{\$}(\mathfrak{M}_s, \mathfrak{M}_t, C, \$)$ *is a constraint model-to-model transformation problem enhanced by an objective function \$. \$ can be a set of objective functions, one for each meta-model in $\mathfrak{M}_t$.*

Proposition 5. *Introducing an objective function \$ for a transformation $T($ $\mathfrak{M}_s, \mathfrak{M}_t, C)$, such that the transformation signature is $T_{\$}(\mathfrak{M}_s, \mathfrak{M}_t, C, \$)$ may only reduce the number of possible solutions for $T(\mathfrak{M}_s, \mathfrak{M}_t, C)$, but an objective function does not affect the number of problems of $T(\mathfrak{M}_s, \mathfrak{M}_t, C)$ which are over-constrained.*

This proposition means that introducing an objective function may only reduce the number of solutions, but never will "destroy" any transformations by reducing the number of solutions to zero. As a result we derive that, from

a theoretical point of view, when trying to reduce the number of possible solutions, introducing objective functions into a transformation definition is always at least as good as sticking without one.

Proof. (Recall that $\|X\|$ counts the number of elements in set X - X may either be a set, a set of models or a number of solutions to a constraint solving problem over models - and therefore may also be applied to transformations) Assume a given input instance model M_s and a transformation $T(\mathfrak{M}_s, \mathfrak{M}_t, C)$, abbreviated T. T is enhanced to $T_\$(\mathfrak{M}_s, \mathfrak{M}_t, C, \$)$, abbreviated $T_\$$, by introducing \$. By definition of optimization problem, if there exists at least one solution, i.e. the transformation is well-defined or under-constraint, then \$ will select a set $M_{t\$}$ of best solutions from M_t. So, we prove part 1 of Proposition 1:

- T is over-constraint $\Leftrightarrow \|M_t\| = 0$. Then \$ cannot select any solution and therefore $\|M_{t\$}\| = 0 \Leftrightarrow T_\$$ is over-constraint.
- T is well-defined $\Leftrightarrow \|M_t\| = 1$. Then \$ selects the only possible solution as the optimal solution and therefore $\|M_{t\$}\| = 1 \Leftrightarrow T_\$$ is well-defined.
- T is under-constraint $\Leftrightarrow \|M_t\| > 1 \Leftrightarrow \|M_t\| \geq 2$. Then there are three cases:

  1. \$ does not influence M_t. Therefore $M_t = M_{t\$} \Rightarrow \|M_t\| = \|M_{t\$}\| \Leftrightarrow T_\$$ is under-constraint.
  2. \$ selects a single, optimal model from M_t. Therefore $M_t \supset M_{t\$} \Leftrightarrow \|M_t\| \geq 2 > 1 = \|M_{t\$}\| \Leftrightarrow T_\$$ is well-defined.
  3. \$ selects a set of optimal models from M_t, but more than one. Therefore $M_t \subseteq M_{t\$} \wedge \|M_{t\$}\| > 1 \Leftrightarrow T_\$$ is under-constraint.

  Since case two is well-defined (but may also be under-constraint) for $T_\$$ and under-constrained for T, we proved, that the number of possible solutions may be reduced for $T_\$$.

Part 2: Assume $T_\$$ under-constraint or well-defined. Then $\|M_{t\$}\| \geq 1$. Then, by definition of optimization problem $\|D\| > 0 \Rightarrow \|M_t\| > 0$. So, in case $T_\$$ under-constraint or well-defined, there are no cases which are over-constraint. Therefore, introducing an objective function does not affect the number of transformations which are over-constraint. □

←

As case two of part one of the proof is of special interest, we derive a new definition:

Definition 6. *An* appropriate objective function *is an objective function that transfers a given under-constraint transformation and set of input models into a well-defined transformation for that set of source models.*

In summary, introducing an objective function may transfer under-constraint transformations into well-defined transformations.

If the transformation is under-constraints several solutions (i.e. interpretations) exist and may be returned by model transformation engines. This will be confusing for developers who will expect that a transformation is interpreted alike using different interpreters. Achieving well defined transformations is therefore highly desirable and should be considered for the case of service composition.

6. Implementation

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

Thereby, the transformation can be executed using the Solverational transformation engine. 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 terms while being executed and re-transformed into EMOF models to provide output to the Eclipse JAVA IDE.

Figure 3 presents some details of the transformation. The alternative domains known from section 5 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 has just been copied over from the formulas presented in Section 3.

Using the results derived in Section 5.1 we derive if the transformation will have an appropriate objective function, which is highly desirable, because the results will be unambiguous. In fact we cannot tell by the objective function

alone whether it is appropriate: if two services will add an identical amount of costs to the result of the objective function there might be ambiguous output models. We therefore must make sure by pairwise comparisons (of the alternative domains given in the transformation) that such two types of services do not exist in our transformation. Using the example below we calculate an upper bound for the number of model elements for which we can be sure that ambiguous results do not exists. We do this by calculating all smallest common multiples of all pairs of costs of types of services. If we know that number, we can calculate the upper bound (by deviding by the largest cost involved in that pair), which is 1850 model elements in our case. We now know that ambiguous results will not be possible below the barrier of 1850 services in a composition. If we will constrain our results to less model elements our objective function can be assumed to be an appropriate objective function.

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

While Solverational might not have the same runtime efficiency as manually coded low-level approaches, it provides a very fast and simple way for developers to generate an implementation at all.

7. State of the Art

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 [4]. However, no universal model-to-model transformation language is used and no optimizations of non-functional properties are made.

Zhu et al. present an approach using UML to describe semantic aspects of web services in OWL-S [10]. They use a model-driven approach to generate Promela descriptions 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 work these models could be used as an abstraction layer to factor out

the mapping from the transformations. However, as Quintero's and Vincente's mappings require a specific tool called OOWS this approach is not solely based on a declarative approach to model transformation.

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

To sum up, Solverational is the only universal declarative model-to-model transformation language which is able to perform optimization and therefore our approach has not been investigated before.

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.

References

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

[2] Jordi Cabot, Robert Clarisó, and Daniel Riera. Verification of UML/OCL Class Diagrams using Constraint Programming. In *Model Driven Engineering, Verification, And Validation: Integrating Verification And Validation in MDE (MoDeVVA 2008)*, 2008.

[3] 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.

[4] 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.

[5] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. OMG, July 2007. ptc/07-07-07.

[6] 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.

[7] Andreas Petter, Miroslav Zlatkov, and Alexander Behring. The solverational grammar and a set of evaluation results. Technical report, Technische Universität Darmstadt, Hochschulstr. 10, 64289 Darmstadt, Jan 2010.

[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*

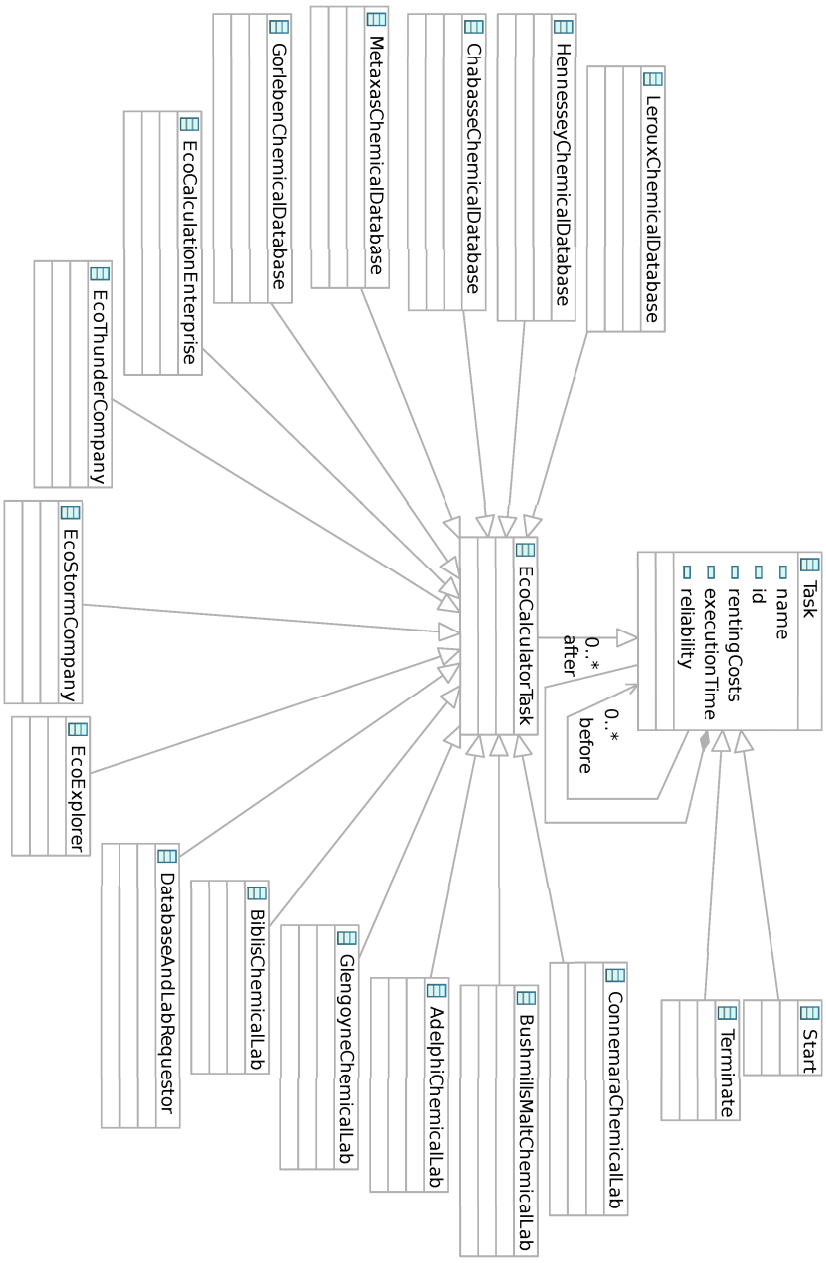

Figure 2: Concrete Meta-Model

```
. . .
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=301
    };
    alternative domain t o:concreteTaskMM::BiblisChemicalLab {
        rentingCosts=150,
        executionTime=2000,
        reliability=300
    };
    alternative domain t
            o:concreteTaskMM::BushmillsMaltChemicalLab {
        rentingCosts=500,
        executionTime=350,
        reliability=301
    };
    . . .
}
. . .
```

Figure 3: Transformation details