Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2011
Online at http://www.jot.fm.

Safe Composition of Transformations

Florian Heidenreich[a] Jan Kopcsek[a] Uwe Aßmann[a]

a. Institut für Software- und Multimediatechnik, Technische Universität Dresden, D-01062, Dresden, Germany

Abstract Model transformations are at the heart of Model-Driven Software Development (MDSD) and, once composed in transformation chains to MDSD processes, allow for the development of complex systems and their automated derivation. While there already exist various approaches to specify and execute such MDSD processes, only few of them draw focus on ensuring the validity of the transformation chains, and thus, safe composition of transformations. In this paper, we present the TraCo composition system, which overcomes these limitations and evaluate and discuss the approach based on two case studies.

Keywords Model Transformation, Safe Composition, Composition Systems

1 Introduction

Model-Driven Software Development (MDSD) [VS06, RS06] is an approach to software development that uses models as its main artefacts where different concerns of the desired software product are described at various levels of abstraction. The overall ideas of MDSD are to transform these abstract models into more concrete models, which can then be used to generate implementation (i.e., source code) or related artefacts and to transform models into other representations on the same level of abstraction. What exactly the models describe and how they are processed by transformations is described in a transformation chain, an MDSD process. This process can for example describe the derivation of a dedicated software product for different platforms—similar to what has been proposed by the Object Management Group (OMG) in its Model-Driven Architecture (MDA) [Obj03]. In Software Product Line Engineering (SPLE) [PBvdL05, CN02], such processes can be used to describe how variability in models is resolved in possibly multiple stages and various transformation steps to create a dedicated product of the Software Product Line (SPL).

MDSD processes usually consist of multiple steps that range from loading and storing of models to performing transformations and generating artefacts. Many of those (e.g., loading models, performing transformations, . . .) are often reused between projects which requires modularisation of the steps into reusable units. Existing work in this direction, e.g. MWE [MWE11] and UniTI [VAB$^+$07], allows for defining and executing custom MDSD processes and reusing common parts across different of such processes.

Although these frameworks provide many benefits for defining MDSD processes, such as stepwise refinement, and are already widely used in industry and academia, they also bear the difficulty of making those transformations work together correctly. The resulting models or implementation artefacts of an MDSD process are only valid if all input models and parameters are valid and each single transformation step receives valid input data and produces valid output data. By validity is thereby meant, the conformance of the input and output data to their corresponding metamodels and their well-formedness rules. Additionally, every transformation step needs to work as intended to ensure validity of the produced output. Because of the many intermediate models it can be very hard to trace a single error in the final product to the input model or the transformation that originally caused that error. Furthermore, there are many heterogenous transformation technologies that are used in combination, suit different use cases and behave differently. This causes additional difficulties when composing multiple steps in such an MDSD process. We observed that existing technologies to describe and execute those processes lack concepts and functionality to ensure the correct interaction between the different transformation steps.

This paper is an extended version of [HKA10] where we present TraCo, a *Tra*nsformation *Co*mposition framework for safe composition of transformations. The goal is to allow the description, composition, and execution of heterogeneous transformation processes, while providing mechanisms for ensuring validity of these processes, which is checked both statically while developing MDSD processes and dynamically while executing them. By composition is meant, the chaining of transformation steps into complex transformations.

The remainder of the paper is structured as follows. Section 2 presents existing work in the area of model transformation and composition of transformation steps. Section 3 presents the conceptual basis for TraCo, introduces the underlying component model and describes the composition of transformation steps. The implementation of TraCo is outlined in Section 4. In Section 5, we present three case studies and discuss the applicability of the approach. Section 6 summarises the paper and presents directions for future work.

2 Background

In this section existing approaches to model transformation are presented. In the scope of this paper, we are interested in approaches that are used to define basic transformation steps (i.e., transformations that are not built by composing multiple, possibly heterogenous, transformations) and composite transformations that are the result of composing basic or composite transformations. First, a short overview of approaches to model transformation is given. Next, we present existing approaches for defining and executing MDSD processes and highlight important criteria that led to the solution presented in this paper.

2.1 Basic Transformation Approaches

In [CH06], Czarnecki and Helsen present an in-depth classification of various transformation approaches regarding multiple criteria. What is visible from this classification is, that there exist a plentitude of approaches, ranging from direct manipulation (e.g., with Java), operational (e.g., QVT Operational [Obj08] or Kermeta [Ker11]), relational (e.g., QVT Relational [Obj08], MTF [IBM04], or AMW [Fab07]), template-based (e.g., model templates [CA05] or FeatureMapper [HKW08]), to approaches based on concepts of graph transformation (e.g., AGG [Tae03] or VIATRA2 [VB07]). All these approaches require specific input models and parameterisation for transformation and provide approach-specific means to constrain this input data. Furthermore, transformations specified using these approaches often lack a formally specified interface which would foster reuse and is key to safe composition of such transformations.

2.2 Composite Transformation Approaches

In the following, we present five approaches for composing basic transformations to composite transformations and highlight important properties of those approaches.

2.2.1 Modeling Workflow Engine

The Modeling Workflow Engine (MWE) [MWE11] is a workflow-based declarative, externally configurable generator engine. An XML-based workflow definition configures the MDSD process by sequentially listing the calls to `WorkflowComponent`s that need to be executed including their parameterisation. openArchitectureWare (oAW) provides simple components for loading and saving models, but also exposes languages for certain tasks through specific components, like validating models using the *Check* language and performing model-to-text transformations using the *Xpand* template language. Each component in a workflow is implemented in a Java class, identified by its class name, and parameterised through one or more tags. The parameter tag names correspond to names of properties of the component class and use its accessor methods. That means that Java classes effectively form the specification and implementation of the components. Using an extended convention for accessor methods, it is easily to distinguish `in`, `out`, and `inout` parameters, work with collection types and perform typing of the parameters against Java types.

Slots that are parameterised by models are not explicitly specified and typed against metamodels. MWE does not provide means to specify constraints for input and output models—although this can be realised using the *Check* language which is similar to the Object Constraint Language (OCL). In order to provide the level of validation desired for safe composition of transformations, *Check* components need to be used before and after each component instantiation. This separates the contract from the component. Putting the constraints and the component into a separate workflow voids the specification of the component. Using a workflow as part of another workflow is also possible. This is done by including the workflow using a reference to the file in which the workflow is defined. It is possible to explicitly define parameter values or simply pass the complete state of the outer workflow to the inner. However, it is not possible to specify the parameters of a workflow explicitly. That means, that knowledge about parameter values given to the inner workflow are solely based on any documentation or comment bundled with the workflow.

2.2.2 UniTI

In [VAB$^+$07], Vanhooff et al. present UniTI, a system specifically designed for defining and executing composed transformation processes. They identify the shortages of existing transformation systems, especially the lack of precise and complete specifications, and provide means to solve these shortcomings. The approach is based on a metamodel that is used to define transformation specifications that can contain multiple input and output parameters. Typing of parameters is done through `ModelingPlatforms` and parameter-specific constraints. Transformation specifications and `ModelingPlatforms` are stored in library elements (`TFLibrary`). In contrast to MWE workflows, UniTI follows a data-driven design. It introduces execution elements for specific transformation steps conforming to a transformation specification, actual parameters (`TFActualParameter`) for every formal parameter and connectors (`Connector`) that connect output and input parameters of different transformation steps. Using this structure it is possible to build non-sequential transformation processes. It is also possible to check whether connections between input and output parameters are valid, because both are typed. Every actual parameter directly links to a model. This model link can either be manually defined—in order to set the input and output models of the transformation chain—or will be automatically set for intermediate models.

Although UniTI provides mechanisms for ensuring the validity of transformation chains (including a precise specification of components, explicit model typing and additional constraints), it lacks an important element. It is not possible to define constraints ranging over multiple parameters, as it is required to express contract conditions (especially if cross-relationships between models exist). So although each single input model can be checked for consistency, the whole input including all models cannot be checked. Similarly, the output cannot be checked for consistency withh the input.

2.2.3 MDA Control Center

In [Kle06], Kleppe presents MDA Control Center (MCC), another approach that supports composition of transformation. Kleppe defines different components that form a composed transformation such as `Creators` (for reading input), `Finishers` (for storing output) and `Transformers`. MCC also provides `ModelTypes` as a means to describe the type of a model. Finally, different means for composing components are provided. First, sequences of two strictly ordered components are possible, where the output of the first is the input of the second. Secondly, parallel combination is allowed, where multiple components take the same input models and the output models are returned as a combined list containing them. Finally, a choice combination is offered that contains multiple components in an ordered list. Every component's condition in this list is tested using the input models and the first components, whose condition is met, is executed or none, if all conditions fail.

Although not further elaborated, MCC does not seem to put more than a reference to the concrete metamodel implementation into the `ModelType` concept. Also, the `ModelTypes` are not directly used in the definitions of the components. They are, however, used in the script language that is used to combine multiple components. This implies that typing of parameters of a component is only done against Java classes. The framework itself does not provide the possibility of defining further constraints on model types. Other conditions such as pre- or postconditions are not available in the specification as well.

2.2.4 AM3

In [ABBJ06], Allilaire et al. present their vision on modelling in the large. Their work is in the context of the AM3 Megamodelling Eclipse project [ATL11] which provides means for resource management (i.e., a repository for models and metamodels), inter-model navigability, and transformation execution. The latter supports transformation execution specification, effective launching of those specifications, means for traceability, and execution recording. Transformation specification is usually done by defining transformations in the ATLAS Transformation Language (ATL) [JAB+06]. Composition of single transformations to composite transformation is supported via a dedicated editor page, where different sub-transformations can be composed to form a `CompositeTransformation`. Furthermore, the approach also supports connectors between transformations that can have multiple values via the `MultiConnector` concept. In [VJBB09], Vignaga et al. present an approach that addresses typing in AM3 and by that, provides means for ensuring type conformance for the artefacts involved in transformations chains.

However, the current status of the work does not allow for the specification of additional constraints that must be fulfilled for the input and output of transformation steps. Also checking the validity of transformation specifications at design time is not supported.

2.2.5 Wires*

In [RRGLR+09], Rivera et al. present Wires*, a system for orchestrating ATL model transformations. They provide a graphical editor for their orchestration language and a corresponding data-flow based engine for the execution of the complex transformations. Wires* provides both sequential and parallel execution of ATL transformations, composition of complex transformations, and offers means for conditional and repeated execution of ATL transformations.

The approach does not yet provide any means for the specification of constraints on the input or output models and is restricted to ATL transformations. As a consequence, similarly to AM3, checking the validity of transformation specifications at design time is not supported.

2.3 Summary

The analysis of existing basic transformation approaches has shown, that many heterogeneous technologies exists and many of them only offer basic means for the specification of external interfaces, their typing, and pre- and postconditions. Only some technologies offer references to actual metamodels and additional constraints for model types. None of the technologies offer explicit contract constraints, that span all input and output parameters.

In the second part of this section, we presented and discussed five concrete approaches for defining and executing composed MDSD processes. In Table 1, we give an overview of important properties of these technologies. All of the technologies provide explicit specifications for the components used in a composed process, although MWE lacks specifications for sub-processes. UniTI, and to a lesser extent MCC, differ from MWE in that they concentrate solely on model transformations. The only parameter type in UniTI is the model parameter, and thus, it is not possible to parameterise components with native values (e.g., paths to input data). MCC allows this for the special components `Creator` and `Finisher` in a specific way. In contrast to that, MWE operates solely on non-model parameter values like strings, numbers, and class references. Models are only addressed through their respective slot name. Where UniTI components cannot receive native values required for parameters, MWE lacks any typing of model parameters. Only UniTI allows more precise specifications using additional constraints, that can be defined to further restrict model types.

The properties of parameters of components used in a composed process also differs between the different technologies. Where UniTI and MCC only support input and output direction, MWE also supports the inout direction. Additionally, they allow multi-valued parameters (which must be supported by the actual transformation technology as well). Again, only UniTI allows to define additional constraints for parameters.

None of the technologies provides the possibility to define contracts for components to ensure the consistency and validity of their input and output data. Despite the fact, that UniTI's constraints for model types and parameters allow to precisely express what kind of model is required, they cannot be used to check consistency across different models. Also, none of the technologies provide additional tool support for design-time validation of MDSD processes in order to identify errors as early as possible in the development process. Besides the metamodel-based specification of components and MDSD processes, we consider contracts for components and extended means for validation as the main requirements for a system that ensures safe composition of transformations.

3 Approach

In this section we conceptualise the framework for safe composition of transformations as a *composition system*. According to [Aßm03], a composition system consists of a *component model*, a *composition language*, and a *composition technique*. The analysis of existing model transformation technologies in Section 2 showed that a key requirement for safe composition of transformation steps is a dedicated *component model*, which allows the complete and precise definition of transformation specifications. In addition, a *composition language* is required, which allows explicit definition of composition recipes and, thus, composed transformation processes. The *composition technique* expands on these concepts by defining how the composition is actually performed. To ensure safe composition, extensive checks for *validation* of those processes need to be performed statically and dynamically.

3.1 Component Model

The component model defines the concepts that can be used to describe arbitrary basic transformation steps that can be used in an MDSD process.

Figure 1 depicts the metamodel of our component model which is defined using Ecore [SBPM08]. It constitutes the actual language to specify components and model types. According to this metamodel, a `Library` consists of multiple `ComponentSpecifications` and `ModelTypes`. A `Library` collects reusable or project-specific `ComponentSpecifications`.

A `ComponentSpecification` has a `name`[1], multiple `PortSpecifications`, `pre-` and `postconditions` and its actual `implementation` definition. `PortSpecifications` have a `name`, a `type` and multiple `Constraints`. The `Type` might have `Con-`

[1]The metaclass `Nameable` was omitted from the metamodel figures to improve clarity.

Table 1 – Overview of transformation composition technologies.

	MWE	**UniTI**	**MCC**	**AM3**	**Wires***
Languages					
For Composition	XML	Model	Textual	Model	Model
For Specification	Java/None for workflows	Component Model	Eclipse Plugins	Model/Eclipse Plugins	Model
Type System					
Native Types	Yes	No	Only internal	No	No
Model Types	No	Yes	Yes	Yes	Yes
Additional Constraints	No	Yes	No	No	No
Parameters					
Supported Directions	All	In, Out	In, Out	In, Out	In, Out
Multi-Value	Yes	No	No	Yes	Yes
Additional Constraints	No	Yes	No	No	No
Validation and Verification					
Contracts for Components	No	No	No	No	No
Design-Time Validation	No	No	No	No	No

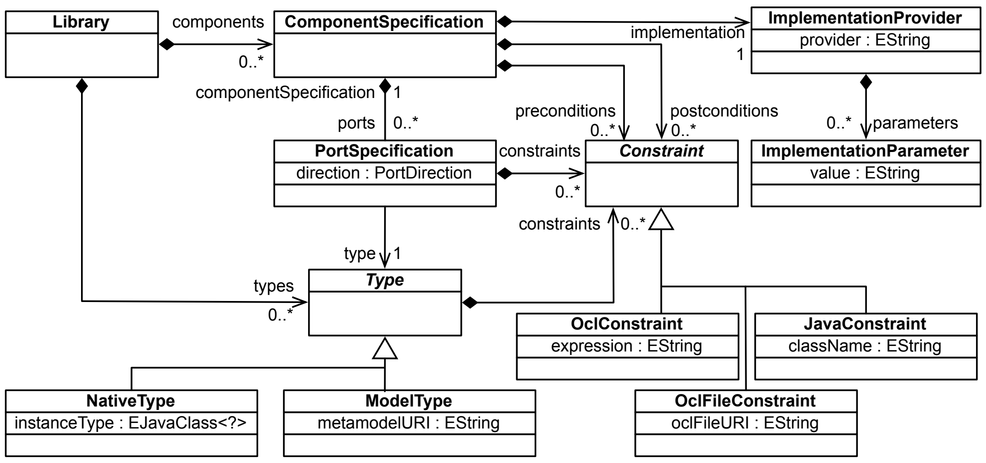

Figure 1 – Metamodel of the Component Model.

`straints` and is either a `ModelType`, which references an EMF-based metamodel declaration, or a `NativeType`, which references a Java type. Currently, no means for subtyping of `ModelType`s are provided.

The pre- and postconditions of a component are realised using `Constraints`. The `OclConstraint` uses a user-defined OCL expression to check the input and output models for consistency. `OclFileConstraint` references a file containing OCL expressions. The `JavaConstraint` metaclass represents constraints implemented in Java. This initial set of constraint types can be easily extended to support other constraint languages.

The `ImplementationProvider` of a component consists of the identifier of its implementation provider and `ImplementationParameters`. The implementation class is not referenced directly, but is made available using a registry of implementation providers. These implementations can be technology-specific adapters or custom component implementations. In contrast to the component implementations, this adapter may enforce a more strict and more well-defined interface on the target classes.

3.2 Composition Language

The presented component model describes the structure of single components. On top of this component model, a dedicated language is provided, which allows the description of composed transformations. This language introduces additional constructs necessary to use and connect components described using the component model.

Figure 2 depicts the metamodel of the composition language. Referenced parts of the component model are coloured grey. According to the metamodel, a `Process` consists of `ComponentInstances`, `ExternalPortInstances` and `Connections`. A `ComponentInstance` is an instance of a component specified using the component model. This instance references its `ComponentSpecification`. Additionally, it may contain `Parameters` to parameterise non-model ports. `ComponentInstances` have `PortInstances` that are instances of the `ComponentSpecification`'s `PortSpecifications`. `PortInstances` and `ExternalPortInstances` can be connected using `Connections`. They may both be the source and target of a `Connection`, although it is not allowed to connect two `ExternalPortInstances`. `ExternalPortInstances` have

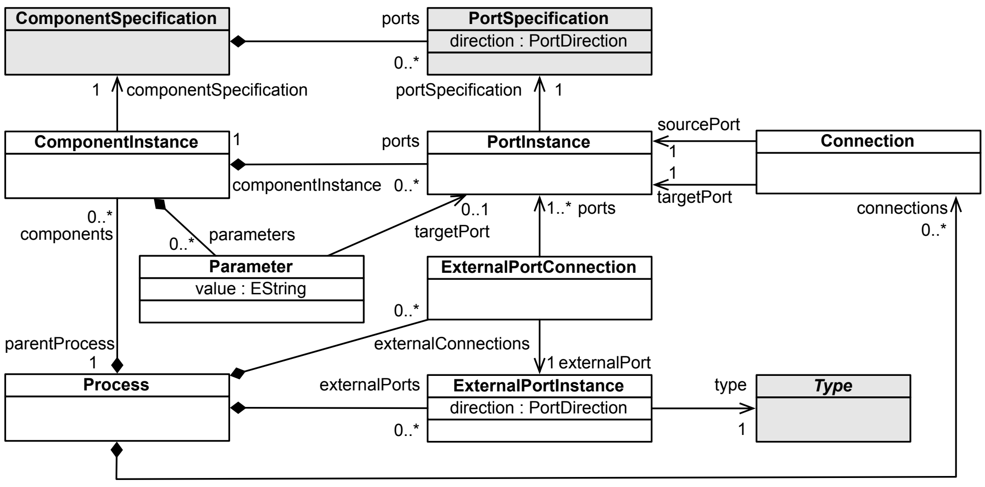

Figure 2 – Metamodel of the Composition Language.

a `name` by which they can be identified and parameterised. The type and direction of the port is inferred from the connection and the opposite port.

Figure 3 depicts an abstract view on transformation components, their interface, and how multiple components are connected to build transformation chains.

3.3 Composition Technique

The composition technique describes the mechanisms necessary to perform the composition of multiple components and execute them. For the developed composition system a black-box composition is used. That means that components only communicate with each other through their ports. The implementation, the implementation parameters and pre- and postconditions are hidden within the component itself and cannot be changed by the composition process.

The execution of a composed transformation process is data-flow based. Any transformation component is executed as soon as it has received data on all its in-ports. In case the component does not have any in-ports, it is considered ready for execution. Any component is only executed once. If there are multiple components ready for execution they are executed in parallel. After a components execution is finished, its output models are transferred to other components via connections. If all components were run exactly once, the process is terminated. Figure 4 depicts the different states in TraCo component execution.

The execution of a transformation component consists of the invocation of its implementation. This implementation may be a technology-specific adapter which initialises and invokes the actual transformation module. After that, the adapter will extract any generated output models from the transformation module and provide them as the component's output models.

In case a component's execution is aborted due to errors or in case it did not generate models for all output ports, the process is aborted. If a component's implementation does not terminate after a global user-defined timeout, this is considered an error as well.

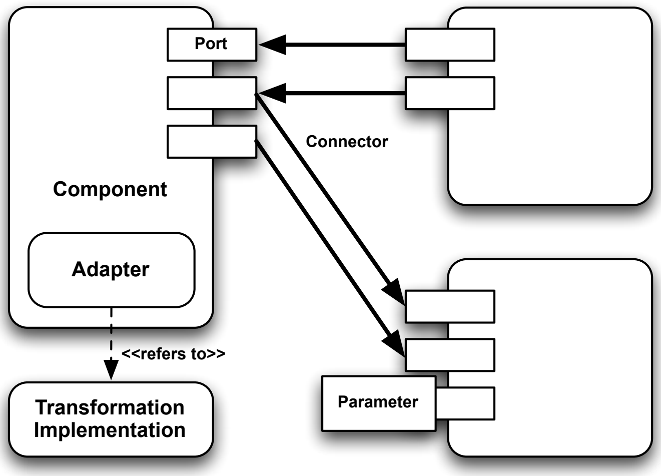

Figure 3 – Transformation components and their composition.

3.4 Validation

Besides the specification and execution of MDSD processes, we want to ensure the consistency and validity of the constructed processes. This is done by performing validation checks for both *intra-component* and *inter-component* consistency.

3.4.1 Intra-Component Consistency

Intra-Component Consistency covers consistency within a single component. This especially includes the implementation of a component, which can consist of a single Java class or an adapter and the actual transformation module. The pre- and post-conditions can also be used to ensure internal consistency to a certain degree. In the following, we describe the various checks performed either statically or dynamically to ensure intra-component consistency.

Consistency of Component Specification Component specifications must be valid and complete. This includes its naming, a reference to an existing component implementation and correct naming and typing of all its port specifications. Additionally, valid port directions have to be specified. This can be checked statically for a single component specification.

Consistency of Implementation Parameterisation If a component implementation requires parameters (e.g., references to resources) to make the actual implementation work, the component implementation has to check the completeness and validity of those parameters. This can be checked statically for a single component specification by the component's implementation.

Consistency Between Specification and Adapted Transformation If a specfific transformation technology is adapted by the framework, the consistency between the component specification and the technology-dependent specification can be checked as well. The level of detail regarding consistency at this level depends on the implementation technology. The more complete and precise the provided specification is, the more checks can be performed, e.g., if the

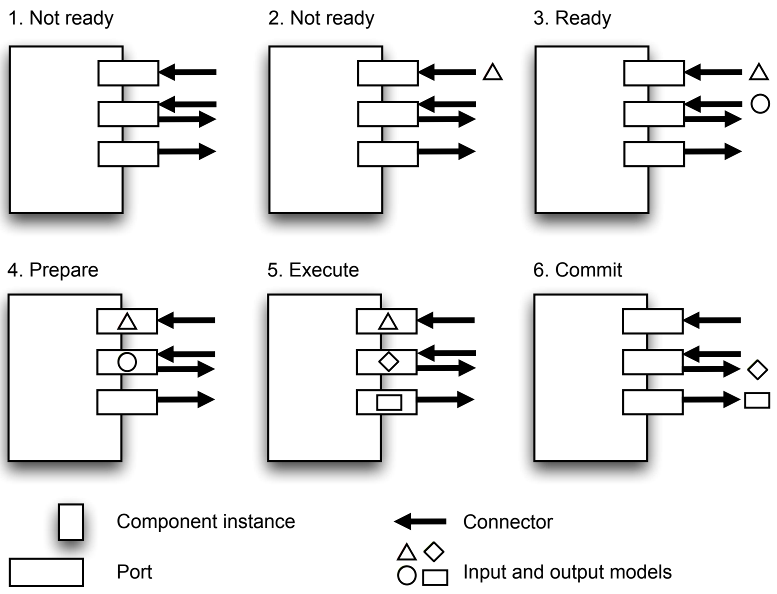

Figure 4 – States in execution of a TraCo component.

technology-depended specification exposes all its required parameters to the outside, the system can check whether all of those are bound to in-ports of the component specification. This can be checked statically for a single component specification by the components implementation.

Consistency of the Transformation Module An adapter implementation can use arbitrary specific transformation technologies. Depending on the concrete technology, there may be means to check the actual implementation for consistency (e.g., checking a transformation script for errors). The level of detail again depends on the amount of information provided. This can be checked statically for a single component specification by the components implementation.

Ensuring Component Contracts The contract of a component, consisting of pre- and postconditions, defines a specific behaviour of the component under certain preconditions. Whether or not this promise holds is hard to check statically against the transformation implementation, because it requires a formal algebra for the underlying implementation technology, which may be Java code and arbitrary transformation technology and may also depend on the concrete input provided. Furthermore, it is difficult to determine whether the contract is violated because of an invalid original input or an erroneous component implementation. At runtime, however, these constraints can be checked against actual input and output models.

Ensuring Implicit Assumptions Even if an MDSD process is completely specified, implicit assumptions on the behaviour of component implementations are drawn. In the given composition system we assume that a component always terminates, produces no errors and outputs data on all output ports. If any of these conditions fail, the process needs to be aborted after a user-defined timeout. This can be checked dynamically on a single component instance by the framework. However, ensuring that a component always terminates cannot be checked by the framework, but the framework can detect whether a component is working longer as expected by the component developer and can report this as error to the user.

Consistency of Type Specifications Native types must have a name and must reference an existing Java type. Model types must also have a name and must reference a valid and existing metamodel. Additional constraints must be valid constraints. This can be checked statically for a single type definition.

3.4.2 Inter-Component Consistency

Inter-Component Consistency covers consistency of the interactions between components within a composed transformation process. All elements including component instances, connectors, parameters, and external ports need to be checked for consistency in different ways. We identified the following checks that ensure inter-component consistency of a composed transformation process.

All Ports Connected Check A component instance must have all its ports connected either by connectors or parameters. For `in` ports, there must be exactly one incoming connection or exactly one parameter. For `out` ports, one outgoing connection and for `inout` ports exactly one incoming connection or parameter and exactly one outgoing connection is required. This can be checked statically for each component instance. Optional ports are currently not supported.

Connection Structure Check It has to be ensured that any connection has a valid port instance as source and as target. These port instances must refer to valid ports of a component specification and must have a valid direction. The source port must have a direction of `out` or `inout`, the target port of `in` or `inout`. This implies that source ports may not be `in` ports and target ports may not be `out` ports. This can be checked statically for each connection.

Connection Type Check This ensures the correct typing between source and target ports of connections which can be checked statically for each connection.

No Cycles Check A cycle is a connection between any of a components output ports to any of its input ports (also across multiple components). Although these compositions are structurally valid, they need to be avoided, as there is no way to invoke any component involved in the cycle due to missing inputs. This can be checked statically for each connection.

Inout Port Type Semantics Check Inout ports have special semantics where it is assumed that the type of data going into this port is the same type of data coming out of it. In order to improve typing for succeeding components it is possible to predict the type of the outgoing data: It will be the same as the type of data going into this port, which was originally the output of the preceding component. It is, thus, possible to check whether the type of the port connected to the outgoing inout port is the same as the type of the port connected to the incoming inout port. This can be checked statically for each inout port.

External Port Connection Check External ports are the connection points of a composed transformation process to other components. They do not perform any action on their own. That is why an external port must be connected to exactly one port of one component. It is not possible to connect two different external ports, without any transformation component in between. Also, an external port must be connected to exactly one component. This can be checked statically for external ports and connections.

Parameters Check The parameters are constant-value connectors that are used to parameterise non-model input ports of a component. Similar to other connections, it has to be checked whether they are connected to a valid port of a component and whether they are correctly typed. This can be checked statically for each parameter.

Runtime Type Check Types of all input and output models can be checked dynamically independent from the component implementation or transformation technology.

Runtime Constraint Check Additional constraints may be defined with the type or port specification. These constraints are checked dynamically for each port when it receives or outputs data.

4 Implementation

Building upon the definition of the composition system, we implemented TraCo as a set of Eclipse plug-ins based on the Eclipse Modeling Framework [SBPM08]. Figure 5 depicts the overall architecture of TraCo. The current implementation consists of tooling to visually build and maintain component libraries and composed transformation processes and an execution environment that performs the actual transformations. These tools also integrate the mechanisms for ensuring intra-component and inter-component consistency as described in Section 3.4. The checks are enforced by the metamodel structure, many of them by dedicated OCL constraints, or programmatic constraints implemented in the component implementations. An example OCL constraint for checking whether each `Connection` has its `sourcePort` and its `targetPort` set is listed below.

```
context Connection inv tracoSourceAndTargetSet:
  not self.sourcePort.componentInstance.oclIsUndefined() and
  not self.targetPort.componentInstance.oclIsUndefined()
```

All errors that are determined statically or dynamically are collected and reported to the developer.

Based on the component model presented in Section 3, an EMF-based editor has been generated and extended by means for easy creation of component specifications (based on wizards) and validation of such specifications. We have also defined a basic set of component specifications and types in the TraCo Standard Library (TSL), which includes components to load and store models and type definitions for common native types and model types.

A visual editor is provided, that allows for specifying MDSD processes by instantiating components from TraCo libraries. Again, this editor performs validation on all elements participating in such a process and reports possible errors to the developer. In addition to the visual editor, TraCo features a textual editor for specification of complex transformation chains. This editor is generated using the textual concrete

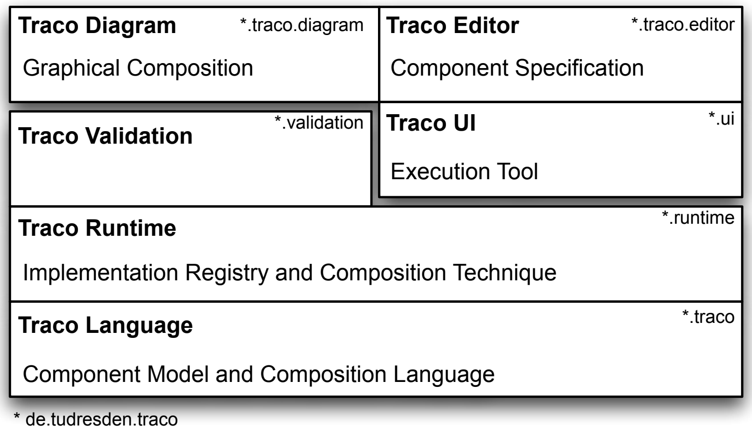

Figure 5 – TraCo Architecture.

syntax tool EMFText [HJK+09] and provides advanced editor features such as syntax highlighting, code completion, quick fixes, and instant error reporting.

A TraCo process can be executed via Eclipse's run configuration dialog. This dialog takes the actual TraCo process as input and the TraCo execution environment performs the invocation of the process and the execution of all referenced components as described in Section 3.3.

Our current experience with TraCo is with using EMF-based models and we designed the tool to fit nicely in the EMF modelling landscape. However, neither the transformed models nor the adapted transformation techniques are limited to the Ecore technological space.

5 Evaluation

We evaluated TraCo based on two case studies performed as student projects, where the students specified transformation components and developed chains of transformations for a given transformation problem. Furthermore, we realised a case study for feature-based customisation of MDSD tool environments using TraCo.

5.1 Case Studies

The first project consisted of the de-facto standard of transformation examples, where a UML class model was transformed to a Relational model. While this project did not provide any new insights regarding the creativity in the field of model transformations, it gave enough motivation to develop the TSL and provided helpful information regarding usability of the tool (ranging from the demand for wizard-based component instantiation to the way how errors are reported to the developer).

The second project was a more complex MDSD scenario where a Java-based issue management system was developed. The students developed multiple EMF-based Domain-Specific Languages (DSLs) that were used to describe the domain model, the user interface, actions and navigation, and the internal application state. Furthermore, this issue management system was designed as an SPL where the focus was more on platform variability than on functional variability (although it also had three

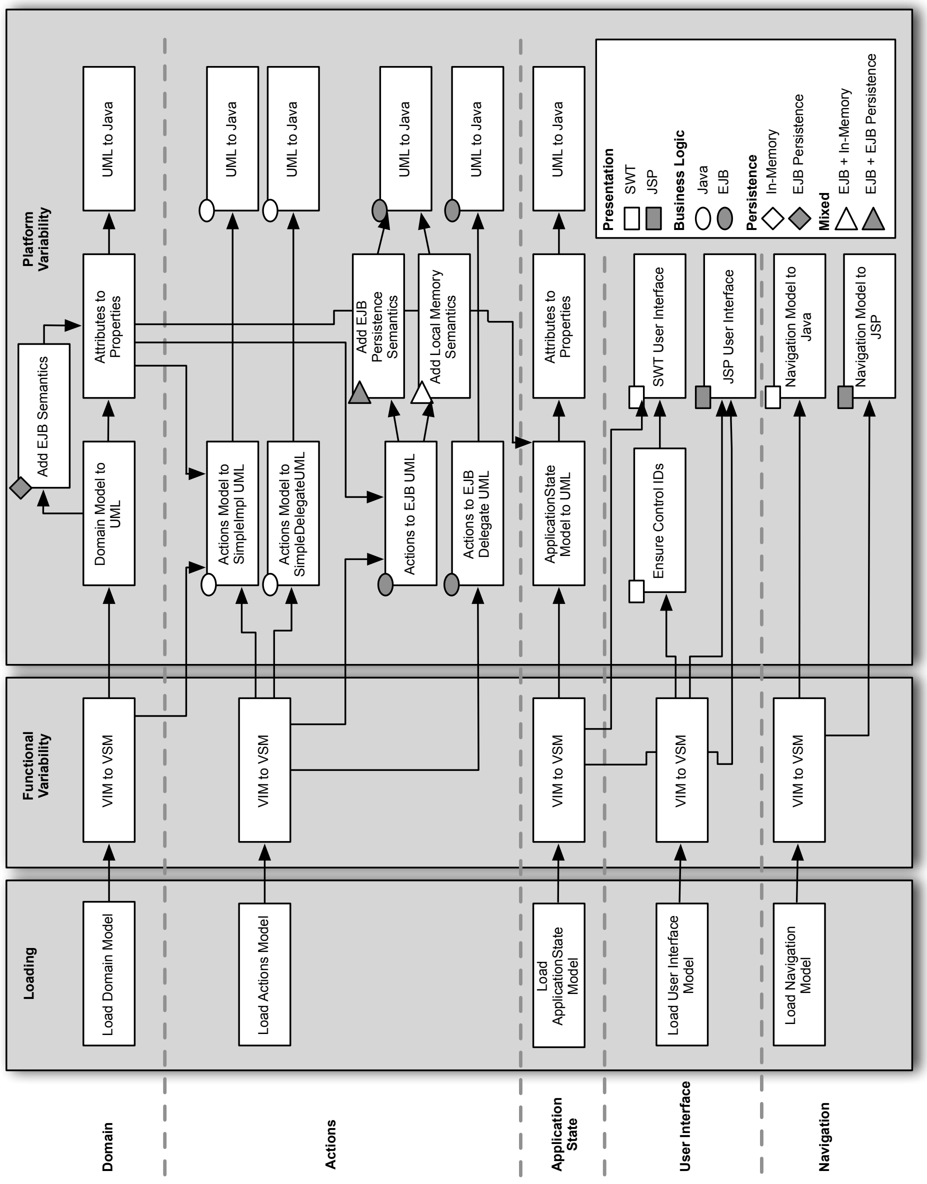

Figure 6 – Example of a two-staged MDSD process.

functional variation points). On platform level, variability between SWT and JSP for the presentation layer, Java and EJB3 on the business layer, and local storage and EJB3 on the persistence layer was required.

We developed a two-staged MDSD process, where platform variability was resolved on the first stage and the actual implementation was generated in various steps on the second stage. To express the variability regarding platform decision, we used our SPL tool FeatureMapper [HKW08, HcW08] and mapped features or combinations of such from a platform variability feature model to the respective transformation component instances and connections of a TraCo process. Figure 6 depicts the overall process where component instances that are used for a specific variant of the software are annotated with a respective shape symbol.[2] To perform the feature-based transformation of the TraCo process in TraCo, we developed a dedicated FeatureMapper component for TraCo which required a TraCo process, a mapping definition and a concrete feature selection as input and produced a TraCo process specific to a concrete platform selection.

This specific process model contained several component instances ranging from loading and storing of models, over feature-based transformation of DSL models and model-to-model transformations with ATL, to model-to-text transformations with MOFScript [Old06].

The feature-based transformation of models was performed on all DSL input models, hence, specifying this transformation as a reusable component was worthwhile. In the overall process, which consists of 32 component instances, this component was reused for each of the 5 DSL models. In addition, this component was also reused for the process on stage one, where the TraCo process itself was transformed and after that executed via a dedicated TraCo component.

In the MOST research project[3], TraCo was used to derive different variants for MDSD tool environments [WZAK10]. In MOST, an MDSD tool environment consists of heterogeneous combinations of different variants of architectural components that form the actual tool environment. These components address software process guidance, support for a specific development method, automation of repetitive tasks, and the ontology technology used. The degree of variability in combining the different variants of the architectural components for tool environments was modelled in a feature model consisting of 28 features. To actually derive a product of this tool product family (i.e., a specific tool environment) TraCo was used in combination with the FeatureMapper SPL tool. Dedicated TraCo components were used to perform the composition of the concrete variant and to perform semantical analysis on the created tool environment.

Figure 7 depicts the derivation process for MDSD tool environments. The process component is parameterised by a feature model by which the variability in tool environments is described. The mapping model relates features or combinations of features from this model to architectural components. The variant model represents a concrete feature selection and, thus, a concrete product of the tool product family. The product derivation component interpretes these parameters and composes a specific MDSD tool environment.

[2]FeatureMapper actually uses colours to visualise the mapping between feature expressions and component instances.

[3]`http://most-project.eu/`

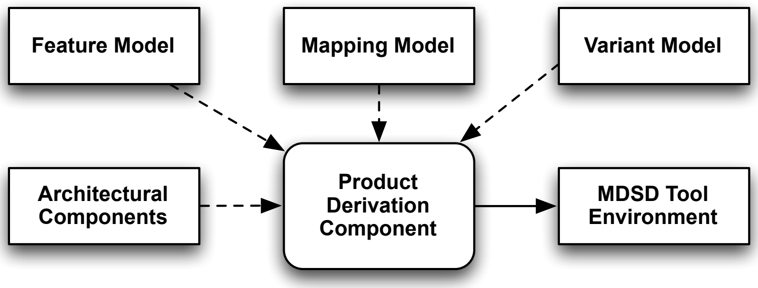

Figure 7 – MDSD Tool Environment derivation process.

5.2 Discussion

The three case studies showed to a certain extent, that it is possible and feasible to create and manage multi-staged MDSD processes with the developed TraCo framework. This does not imply that this observation can be directly generalised to other development efforts. However, as we will see, a number of observations reflect the fact that improved means for validation of complex transformation processes result in safer transformation chains, so there is some tentative grounds for careful generalisation.

The composition system and its supplemental tools made it possible to incorporate heterogeneous transformation technologies including ATL, MOFScript, and feature-based transformation using FeatureMapper. The validation features on component-specification level helped at identifying and fixing erroneous specifications. This also included the detection of component specifications that mismatch with their corresponding transformation modules (which was the case when transformation modules were refactored).

Similarily, for the transformation processes, the extended means for validation proved to be helpful at identifying and fixing type errors, yet unconnected ports or ports that did no longer exist because of changes in the component specification. The definition of constraints and contracts using OCL enabled us to more precisely specify expected input and output data. As an example, the transformation for the SWT user interface required unique identifiers on model elements, but the DSL itself did not enforce this. By defining an additional transformation to augment model elements with generated identifiers and providing an additional precondition to the model transformation, we were able to create a reusable transformation component without putting more bloat to the original transformation—while still maintaining integrity of the whole transformation.

The preconditions also helped when working with multiple input models in transformations. For example, some transformations rely on the consistency between two given input models. Any state variable referenced in the first model (in this case the user interface model) must be declared in the second model (in this case the application state model). Similarly, the transformations of action models require consistency of references to state variables as well. Especially in the context of multiple possible variants of the source models—as a result of feature-based transformation—the preconditions help to quickly identify consistency errors. We observed, that the reasons for these errors were not always inconsistent models, but also incorrect mappings of features to DSL model elements. In addition, we also noticed another issue when working with multiple variants specified in one model. Some constraints can no longer be applied without taking the mapping information into account, e.g., when components have multiple incoming connections for the same port which is not allowed in a single-variant scenario and detected by TraCo's validation mechanisms. They can, however, be used to ensure consistency of a specific process variant.

6 Conclusion

In this paper, we presented TraCo, a framework for *safe* composition of transformations. First, we analysed existing work in this field and derived requirements for TraCo from our observations. In addition to metamodel-based specification of components and MDSD processes, we identified contracts for components and means for validation of component specifications and MDSD processes as the main requirements for a system that ensures safe composition of transformations. Second, we conceptualised a composition system (consisting of the modules component model, composition language and composition technique) for TraCo and presented a classification of checks that can be performed to ensure intra-component and and inter-component consistency, thus, resulting in a system that allows for safe composition of transformations. We outlined TraCo's implementation and presented three case studies, followed by a discussion of the applicability of the approach.

In the future, we want to further evaluate TraCo and possibly integrate the concepts developed in this work with existing approaches, preferably MWE. On implementation level, means for debugging TraCo processes are planned.

References

[ABBJ06] Freddy Allilaire, Jean Bézivin, Hugo Brunelière, and Frédéric Jouault. Global Model Management in Eclipse GMT/AM3. In *Proceedings of the Eclipse Technology eXchange Workshop (eTX), collocated with the 20th European Conference on Object-Oriented Programming (ECOOP 2006)*, 2006.

[Aßm03] Uwe Aßmann. *Invasive Software Composition*. Springer, 2003.

[ATL11] ATLAS Group. AtlanMod MegaModel Management (AM3), 2011. URL `http://eclipse.org/gmt/am3/`.

[CA05] Krzysztof Czarnecki and Michał Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *Proceedings of the 4th International Conference on Generative Programming and Component Engineering (GPCE 2005)*, pages 422–437, 2005. `doi:10.1007/11561347_28`.

[CH06] Krzysztof Czarnecki and Simon Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal – Special Issue on Model-Driven Software Development*, 45(3):621–646, 2006. `doi:10.1147/sj.453.0621`.

[CN02] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[Fab07] Marcos Didonet Del Fabro. *Metadata Management Using Model Weaving and Model Transformation*. PhD thesis, University of Nantes, 2007.

[HcW08] Florian Heidenreich, Ilie Şavga, and Christian Wende. On Controlled Visualisations in Software Product Line Engineering. In *Proceedings of the 2nd Workshop on Visualisation in Software Product Line Engineering, collocated with the 12th International Software Product Line Conference (SPLC 2008)*, September 2008.

[HJK$^+$09] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and Refinement of Textual Syntax for Models. In Richard F. Paige, Alan Hartman, and Arend Rensink, editors, *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2009)*, volume 5562 of *Lecture Notes in Computer Science*, pages 114–129. Springer, 2009. doi:10.1007/978-3-642-02674-4_9.

[HKA10] Florian Heidenreich, Jan Kopcsek, and Uwe Aßmann. Safe Composition of Transformations. In Laurence Tratt and Martin Gogolla, editors, *Proceedings of the 3rd International Conference on Theory and Practice of Model Transformations (ICMT 2010)*, volume 6142 of *Lecture Notes in Computer Science*, pages 108–122. Springer, June 2010. doi:10.1007/978-3-642-13688-7_8.

[HKW08] Florian Heidenreich, Jan Kopcsek, and Christian Wende. FeatureMapper: Mapping Features to Models. In *Companion Proceedings of the 30th International Conference on Software Engineering (ICSE 2008)*, pages 943–944, New York, NY, USA, 2008. ACM. doi:10.1145/1370175.1370199.

[IBM04] IBM United Kingdom Labratories Ltd., IBM alphaWorks. Model Transformation Framework (MTF), 2004. URL http://www.alphaworks.ibm.com/tech/mtf.

[JAB$^+$06] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. ATL: a QVT-like transformation language. In Peri L. Tarr and William R. Cook, editors, *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2006)*, pages 719–720. ACM, 2006. doi:10.1145/1176617.1176691.

[Ker11] Kermeta Project Team. Kermeta, 2011. URL http://www.kermeta.org/.

[Kle06] Anneke Kleppe. MCC: A Model Transformation Environment. In *Proceedings of the 2nd European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2006)*, volume 4066 of *Lecture Notes in Computer Science*, pages 173–187. Springer, 2006. doi:10.1007/11787044_14.

[MWE11] MWE Project Team. Modeling Workflow Engine, 2011. URL http://www.eclipse.org/modeling/emft/?project=mwe.

[Obj03] Object Management Group. MDA Guide Version 1.0.1. OMG Document, 2003. URL http://www.omg.org/cgi-bin/doc?omg/03-06-01.

[Obj08] Object Management Group. MOF QVT Specification, v1.0. OMG Document, 2008. URL http://www.omg.org/spec/QVT/1.0/.

[Old06] Jon Oldevik. MOFScript User Guide, 2006. URL http://www.eclipse.org/gmt/mofscript/doc/MOFScript-User-Guide.pdf.

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.

[RRGLR+09] José E. Rivera, Daniel Ruiz-Gonzalez, Fernando Lopez-Romero, José Bautista, and Antonio Vallecillo. Orchestrating ATL Model Transformations. In *Proceedings of the 1st International Workshop on Model Transformation with ATL (MtATL 2009)*, pages 34–46, 2009.

[RS06] John J. Ritsko and David I. Seidman. Preface. *IBM Systems Journal – Special Issue on Model-Driven Software Development*, 45(3), 2006. doi:10.1147/sj.453.0449.

[SBPM08] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *Eclipse Modeling Framework, 2nd Edition*. Pearson Education, 2008.

[Tae03] Gabriele Taentzer. AGG: A Graph Transformation Environment for System Modeling and Validation. In *Tool Exhibition at Formal Methods 2003*, 2003.

[VAB+07] Bert Vanhooff, Dhouha Ayed, Stefan Van Baelen, Wouter Joosen, and Yolande Berbers. UniTI: A Unified Transformation Infrastructure. In *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems (MODELS 2007)*, volume 4735 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2007. doi:10.1007/978-3-540-75209-7_3.

[VB07] Dániel Varró and András Balogh. The Model Transformation Language of the VIATRA2 Framework. *Science of Computer Programming – Special Issue on Model Transformation*, 68(3):214–234, 2007. doi:10.1016/j.scico.2007.05.004.

[VJBB09] Andrés Vignaga, Frédéric Jouault, María Bastarrica, and Hugo Brunelière. Typing in Model Management. In Richard F. Paige, editor, *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations (ICMT 2009)*, volume 5563 of *Lecture Notes in Computer Science*, pages 197–212. Springer, 2009. doi:10.1007/978-3-642-02408-5_14.

[VS06] Markus Völter and Thomas Stahl. *Model-Driven Software Development*. John Wiley & Sons, 2006.

[WZAK10] Christian Wende, Srdjan Zivkovic, Uwe Aßmann, and Harald Kühn. Feature-based Customisation of MDSD Tool Environments. Technical Report TUD-FI10-05-Juli 2010, Technische Universität Dresden, 2010. Available from: ftp://ftp.inf.tu-dresden.de//berichte/tud10-05.pdf.

Acknowledgments This research has been partly co-funded by the German Ministry of Education and Research within the project feasiPLe.