Designing Information Hiding Modularity for Model Transformation Languages

Andreas Rentschler Dominik Werle Qais Noorshams Lucia Happe Ralf Reussner

Karlsruhe Institute of Technology (KIT)
76131 Karlsruhe, Germany
{rentschler, noorshams, happe, reussner}@kit.edu
dominik.werle@student.kit.edu

Abstract

Development and maintenance of model transformations make up a substantial share of the lifecycle costs of software products that rely on model-driven techniques. In particular large and heterogeneous models lead to poorly understandable transformation code due to missing language concepts to master complexity. At the present time, there exists no module concept for model transformation languages that allows programmers to control information hiding and strictly declare model and code dependencies at module interfaces. Yet only then can we break down transformation logic into smaller parts, so that each part owns a clear interface for separating concerns. In this paper, we propose a module concept suitable for model transformation engineering. We formalize our concept based on *cQVTom*, a compact subset of the transformation language QVT-Operational. To meet the special demands of transformations, module interfaces give control over both model and code accessibility. We also implemented the approach for validation. In a case study, we examined the effort required to carry out two typical maintenance tasks on a real-world transformation. We are able to attest a significant reduction of effort, thereby demonstrating the practical effects of a thorough interface concept on the maintainability of model transformations.

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Design Tools and Techniques—Modules and Interfaces; D.3.2 [*Programming Languages*]: Language Classifications—Specialized Application Languages

General Terms Design, Languages

Keywords Model-driven software engineering, model transformations, transformation languages, modularity, maintenance

1. Introduction

In software engineering circles, domain-specific languages (DSLs) have gained wide acceptance as a technique to improve the productivity and quality of software. This is particularly true for model-driven software engineering where models represent first-class artifacts. In this field, a multitude of specialized languages had been designed to transform models into other models and finally into code artifacts. These so-called transformation languages aim to ease development efforts by offering a succinct syntax to query from and map model elements between different modeling domains.

However, development and maintenance of model transformations themselves are expected to make up a substantial share of the lifecycle costs of software products that rely on model-driven techniques [20]. Much of the effort in understanding model transformations arises from complexity induced by a high degree of data and control dependencies. Complexity is connected to size, structural complexity and heterogeneity of models involved in a transformation. Visualization techniques can help to master this complexity [17, 20]. It is a fact that maintainability must already be promoted at the design-time of a software program [18]. As one solution to ease maintenance processes, modern programming languages feature concepts to decompose programs into modules or classes. In addition, the concept of interfaces helps to hide implementation details, thus making it easier for developers to understand a program, to locate concerns, and to adapt a program to new or changing requirements. These concepts reach back to the 1970s when Parnas proposed the information hiding principle as a key principle for software design [16]. Nowadays, they are well-accepted for their positive effect on maintainability [13].

At the present time, however, there exists no concept for model transformation languages that allows programmers to control information hiding and strictly declare model and code dependencies at module interfaces. Yet only then can we break down transformation logic into smaller parts, where each part owns a clear interface for separating concerns. To the same extent as for programs written in general-purpose languages, with a proper concept to encapsulate concerns, the effort required in understanding behavior and locating concerns in larger transformations can be significantly reduced.

Among the many DSLs that had been designed for model transformation programming, QVT, ETL, and ATL belong to the most popular and advanced ones. But all these and other proposed transformation languages lack a thorough module concept. QVT's concept of libraries, for example, does only allow to dissect a transformation into smaller parts. There are no explicit interfaces, and it is impossible to hide functionality. Thus, in order to understand a QVT transformation, it is necessary to read the full implementation.

What distinguishes transformation programs from general-purpose programs is that they operate on often large and structurally complex models. However, existing transformation languages merely provide weak encapsulation mechanisms: developers cannot specify what model elements a module is allowed to read, instantiate, or modify. By just looking at the interface of a module, one cannot tell its impact. But, as long as interfaces are not able to communicate on which of the models' elements a module operates, developers

Copyright © ACM, 2014. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in Designing Information Hiding Modularity for Model Transformation Languages, Proceedings of the 13th international conference on Modularity, http://dx.doi.org/10.1145/2584469.

MODULARITY '14, April 22–26, 2014, Lugano, Switzerland.
Copyright © 2014 ACM 978-1-4503-2772-5/14/04...$15.00.
http://dx.doi.org/10.1145/2577080.2577094

are struggling with understanding a module's impact on the overall transformation.

In this paper, we propose a module system that includes not only control dependencies as part of its interface contracts, but also data dependencies at the class-level of involved models. We formalize our approach by designing a minimal yet fully functional transformation language, Core QVT-Operational-Modular, short *cQVTom*, that is based on the conceptual core of QVT-Operational (QVTo) and that embraces an interface for rigorous specification of both control and data dependencies. The goal of our approach is to attest similar effects on model transformations as observed for modularized programs in general, namely improved understandability, maintainability and adaptability [18].

As a proof-of-concept and to validate the expected effects of our approach, we integrate our modular concept into the transformation language Xtend. We chose Xtend so we can reuse Java's interface and class concepts provided by Xtend's host language, Java, and adapt it to our needs. We carry out a case study on a real-world model-to-text transformation written in Xtend. For two typical maintenance scenarios, a refactoring and an evolution scenario, we can demonstrate how our approach helps to localize concerns already at the interface-level without examining the underlying implementation, thereby significantly reducing the effort as opposed to a previous, non-modularized version.

In summary, we make two different contributions: Firstly, we design a proper module concept for model transformation languages, which we formally define on a core subset of QVTo. Secondly, we validate our approach on a real-world model-to-text transformation written in Xtend. For this purpose, we implemented our concept for the Xtend language.

This paper is structured as follows. First, we motivate the difficulty of maintaining transformations in Section 2, and we present our idea in Section 3. In Section 4, we formalize the approach by providing syntax and a type system for a core subset of QVTo that is enriched with our module concept. Section 5 introduces our implementation into the Xtend language, and Section 6 studies effects of our approach on maintainability of a real-world transformation. In Section 7, related work is discussed. Finally, Section 8 presents conclusions and proposes directions for future work.

2. Maintenance of Model Transformations

In model-driven software engineering, models are considered to be first-class artifacts and are expected to evolve during their lifecycle. Whenever a model changes, all dependent artifacts of the model must be adapted for that change, including model instances and transformations that operate on the model (co-evolution). It is thus important that model transformations can be adapted with minimal effort. This effort can be minimized through a modular design, under the premise that the programming language supports adequate concepts for modularization.

By means of two realistic evolution scenarios, we identify issues that appear with existing module concepts of transformation languages. We chose an example transformation that is small enough to help clarify these issues while still presenting the core features of an imperative transformation language. Let us consider a unidirectional transformation between two similar models, one model describes simple activity diagrams, the other describes processes as chains of steps. Figure 1 uses a textual syntax that is part of the QVTo language [14] to define both models.

A transformation from activity to process models requires two mappings to be implemented. One mapping creates for each instance of `Activity` an instance of class `Process`, and another mapping is responsible for projecting any `Action` contained in an `Activity` to a `Step`, so that the `Step` is contained in the `Process` that had been created from the respective `Activity`. Since we

```
package ActivityModel {
  class Activity { composes actions : Action [*]; }
  class Action { references succ : Action [1]; }
  class StartAction extends Action { }
  class StopAction extends Action { }
}
package ProcessModel {
  class Process { composes steps : Step [*]; }
  class Step {
    references next : Step [1];
    composes isStart : Boolean [1];
    composes isStop : Boolean [1];
  }
}
```

Figure 1: Activity2Process example – Source and target models

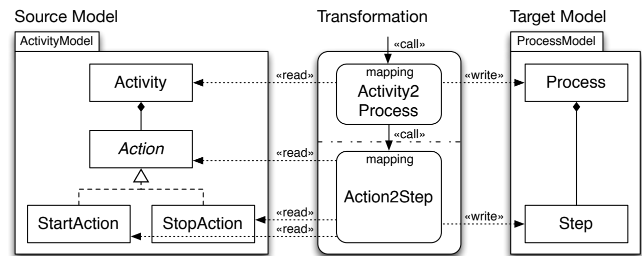

Figure 2: Activity2Process transformation

are anticipating that in the future both models might be extended with new entities, and that the exact behavior of these mappings might be subject to change, we find it reasonable to encapsulate each mapping in a separate module. Figure 2 depicts model and mapping dependencies that occur.

In QVTo, both mappings can be implemented as mapping operations and contained in two separate modules, a transformation and a library module. Because of QVTo's imperative nature, the transformation module must contain an entry point, the main function. The main function's signature declares input and output models as transformation parameters with a model type, also called the domains of a transformation. An implementation of the transformation module is given in Figure 3a. The latter mapping operation is factored out into an own module that is imported by the main module (Figure 3b).

Several weaknesses of this approach become evident when we consider two typical maintenance scenarios, refactoring and evolution. Failure to perform refactoring at periodic intervals usually results in accumulating technical debt. Evolution of transformations can be experienced even more frequently, emerging whenever any of the models change.

Refactoring modular structure. Suppose we want to rethink the way the transformation is currently structured. In our example, module `Activity2ProcessModule` is responsible for mapping an `Activity` to a `Process`, and module `Action2StepModule` for mapping instances of `Action` to instances of `Step`. Dependence information is only available by studying the actual implementation. Module clustering is generally determined by dependence metrics like coupling. In contrast to ordinary software, external dependencies are not only introduced by method calls, but also by references to model element. Without having an interface concept that incorporates both dimensions, control and data flow, it is hard to reason about modularity of model transformations. Furthermore, the subset of model elements a module does access is only by convention, it is neither programmatically declared nor automatically enforced.

Adapting to evolving models. Adding an attribute `name` to actions and steps requires the transformation to adapt accordingly

```
import Action2StepModule;
transformation Activity2ProcessModule(
  in a:ActivityModel, out p:ProcessModel)
  extends Action2StepModule;
main() {
   a.rootObjects()[Activity]->map mapActivity2Process();
}
mapping Activity::mapActivity2Process() : Process {
   result.steps := self.actions->map mapAction2Step();
}
```

(a) First module

```
transformation Action2StepModule(
  in a:ActivityModel, out p:ProcessModel);
mapping Action::mapAction2Step() : Step {
   -- result.name := self.name;
   result.next := self.succ.late resolveone(Step);
   result.isStart := self.oclIsTypeOf(StartAction);
   result.isStop := self.oclIsTypeOf(StopAction);
}
```

(b) Second module

Figure 3: Activity2Process example in QVTo

(Figure 3b, line 4). We cannot deduce from the transformation signature alone for what model parts each module is responsible. Because mapping implementations might internally access classes `Action` or `Step`, their signatures are neither informative enough. Developers need to check the full code to locate relevant spots before carrying out the requested change.

Other transformation languages, like ATL, ETL, Kermeta, and VIATRA2, provide even less sophisticated concepts than QVTo. In a classification from 2003 [6], Czarnecki observes that only some approaches support organization of rules into modules, he does not talk about information hiding aspects. Transformation languages that are hosted by an object-oriented language may exploit the available class mechanism, for example RubyTL for Ruby, SMTL for Scala, and Xtend for Java. But even Java's sophisticated class mechanism still does not offer a solution for model access control.

3. Modularity Tailored for Transformations

To solve mentioned issues, our idea is to introduce a proper interface concept that facilitates information hiding. Hiding the internal details of a piece of software from any other piece and encapsulating design decisions that are likely to change bring the following benefits (cf. MacCormack [13], Parnas [16]):

Encouraging deliberate designs. Implementations are type-checked against their interfaces to ensure contracts are met. This encourages developers to think about a modular design with encapsulated implementation decisions. It also prevents misuse by introducing unintended dependencies from external code.

Fostering team development. Initial development can be carried out more efficiently with a proper interface concept. As soon as a modular design has been created in terms of interfaces, multiple developers may work on implementing different parts of the project in parallel.

Making software easier to understand, use and reuse. Developers require less effort to understand software behavior when provided with a view that abstracts from implementation details. Interfaces offer an abstract and often sufficient description of a module's responsibilities and dependencies.

Simplifying modification and repair. If a decision is distributed over multiple modules, ripple-of-change effects occur when that decision is being modified. Localized design decisions help to limit ripple-of-change effect.

Facilitating variability through reconfiguration. Alternative implementations can be easily exchanged for design decisions that are encapsulated behind an interface. This is done by simply exchanging implementations of the same interface.

So that developers can exploit these benefits to the fullest, we expect a model transformation language to integrate a module concept that abides to the information hiding principle. Our understanding of the principle is described by these four rules:

R1: Segregation of interface and implementation. Implementation details (queries and helper methods, internal state) are hidden behind interfaces. Per required interface, a unique implementation exists.

R2: Conformance of interface and implementation. Implemented methods must conform to exported method signatures. This means that method name and the number of arguments have to be equal, and types must be substitutable according to Liskov (contravariance of argument types, covariance of return types). Any method used must be either defined locally in the implementation, or it must be defined in one of the imported interfaces.

R3: Method access control. Only methods that are either defined locally or by an imported interface are visible. Any other method is not visible and can therefore not be accessed.

R4: Model access control. Here, our domain-specific module concept is distinct from module concepts for general-purpose programming languages. In order to accommodate the central role of model references for transformation languages, interfaces must make the scope of model elements in a domain that can be referenced explicit. An implementation can access only model elements that are defined by exported interfaces. For greater flexibility, access restrictions on models should be definable not only at class-level but also at package-level. The latter is equivalent to explicitly stating any transitively nested classes.

Violation of interface contracts should be detected at design-time employing static type checking. A compile-time error should be issued if methods that are hidden are accessed, if model elements are referenced that are not declared as modifiable or readable, or if methods that are declared in an interface remain unimplemented or with incompatible signatures.

We propose a derivative of QVTo, we name it QVT-Operational-Modular (QVTom), that replaces the existing module concept with a more elaborate one. To give an idea of the notation, we rewrite the previous example in QVTom (cf. Figure 4). A `transformation module` must implement at least one interface and can depend on an arbitrary number of interfaces, stated by keywords `export` and `import`, respectively. A `transformation interface` must be exported by exactly one module implementation. An interface declares a transformation signature, which is a list of typed model parameters. Signatures of an implementation's exported interfaces must be identical, except for access restrictions. For each model parameter, directives `in`/`out`/`inout` indicate the direction, and access is restricted to the classes and packages listed in trailing square brackets `[]`. A package name is a shortcut for any directly or indirectly contained classes in the package. Any reference to a model element is put into the context of a model parameter, e.g. `Activity@a`. Modules must define at least the methods declared in an exported interface with compatible signature.

There has to be one dedicated interface `IMain` with exactly one mapping that forms the entry point of a transformation. This approach is borrowed from the Modula-3 language. In our example, module `Activity2Process` exports `IMain` (Figure 4a), so that mapping `mapActivity2Process` is the entry point.This mapping calls another mapping provided by interface `IAction2-Step`, and implemented by module `Action2Step` (Figure 4b).

```
transformation interface IMain(
    in a : ActivityModel[Activity],
    out p : ProcessModel[Process]) {
  mapping Activity@a::mapActivity2Process() : Process@p;
}
transformation module Activity2Process
    export IMain
    import IAction2Step {
  mapping Activity@a::mapActivity2Process() : Process@p {
      result.steps := self.actions->
        map IAction2Step::mapAction2Step();
  }
}
```

(a) First module

```
transformation interface IAction2Step(
    in a : ActivityModel[StartAction, StopAction],
    out p : ProcessModel[Step]) {
  mapping Action@a::mapAction2Step() : Step@p;
}
transformation module Action2Step
    export IAction2Step {
  mapping Action@a::mapAction2Step() : Step@p {
     result.name := self.name;
     result.next := self.succ.late resolveone(Step@p);
     result.isStart := self.oclIsTypeOf(StartAction@a);
     result.isStop := self.oclIsTypeOf(StopAction@a);
  }
}
```

(b) Second module

Figure 4: Activity2Process example in QVTom

Implementations of each interface are granted restricted read or read/write access to distinct subsets of the models, whereby access to a class automatically implicates access to the class's features. In the example, implementations of `IMain` can only access instances of `Activity`, and create or modify instances of `Process`. On the other hand, implementations of `IAction2Step` can only refer to instances of `StartAction` and `StopAction`, and create or modify objects of class `Step`. If the latter module implementation would define further queries or mappings, these would not be visible to the former module's implementation.

The purpose of a module system is namespace control and data abstraction. There are no extra semantics added besides definition and resolution of namespaces, access control on top of namespaces, and an aligned mechanism for entry point definition. Thus, QVTom programs can always be transformed into non-modular QVTom or QVTo programs by giving unique names to entities.

Revisiting both example scenarios from last section, we can easily see that the proposed module system brings certain benefits. Refactoring the modular structure requires less effort, as all of the information required for reasoning can be deducted from the interface definitions alone. When it comes to adapting the example to an evolving model, we can locate the affected module much quicker from reading the interface descriptions as well: only one of the two modules has access to subclasses of `Action`.

4. Core QVTom

In the style of Featherweight Java (FJ) [10] we formalize a minimal subset of QVTom that we call *Core QVTom* (*cQVTom*). Main purpose of cQVTom is to demonstrate the added modular system. While retaining core features of transformation languages, we skip several of QVTo's features that do not add to the general idea and should be integrable straightforwardly. In this section, based on the syntax, we present a calculus for type inference and prove its soundness. We guarantee that a well-typed program enforces information hiding postulated by the four rules from the previous section.

Core QVTom skips several metamodeling concepts (e.g., abstract classes, primitive types, multiplicities), many of QVTo's concepts (e.g., helpers, queries, constructors, variables and globals, superimposition, dispatching, guards and sections), and most concepts of the underlying *Object Constraint Language* (if, let, collection operations, stdlib functions). QVT's existing module concepts had been completely removed to be replaced by our concepts, e.g., import, access, extend, transform, main.

We realize modularity as a second-class module system. This means that the module system is segregated from the core language's system. Modular definitions are evaluated statically at compile-time, hence at runtime, expressions cannot reflect on modules nor can they manipulate them. If a program is well-typed it conforms to the information hiding rules. Later at runtime, the module structure can be ignored as it is no longer needed[1].

4.1 Syntax

The syntax is minimal, though expressive enough to demonstrate relevant features and interaction of the added features together with core features of QVTo. The abstract syntax is presented in Figure 5 using a variant of the Backus-Naur form. A transformation T comprises three parts, metamodel definitions, interface definitions, and module implementations, the latter defining mapping implementations with a minimal QVTo syntax.

Metavariables p, c, f, i, t, s, x range over unique names of packages, classes, fields, interfaces, domains, mappings, and both arguments and variables, respectively. Typing judgments on sequences are abbreviated, $\overline{P}$ is shorthand for whitespace-separated lists $P_1 \ldots P_n$ with zero or a finite number of elements, analogous for $\overline{C}$, $\overline{F}$, $\overline{S}$, $\overline{O}$, $\overline{B}$. In order to hint the underlying concrete syntax, the overline operator with an overset comma denotes shorthand notation for comma-separated parameter lists, e.g., $\overset{,}{\overline{e}}$. The ? operator marks grammatical expressions as optional, the | operator separates alternative choices.

Metamodels are formulated with the same notation as described in the QVT specification, with one extension: Packages p not only define classes, they define subpackages as well. A class c can inherit from another class c', and define contained or referenced elements. A field f is typed with a class c and can have multiplicities 1 or $0..*$.

A module interface i specifies a list of model domains $\overline{t : p}$ the transformation unit is operating on, where a domain either acts as input or output (`in` or `out`). A domain owns a unique model domain identifier t that is part of any model element reference. In addition to root package p, the exact elements contained in the root package that are accessible by implementations must be declared in trailing square brackets. This can be a list of classes $\overline{c'}$ and packages $\overline{p'}$. Naming a package equals to naming all directly and indirectly contained classes in the package. Method signatures of mappings are identical to QVTo's syntax. Each metamodel element, the calling context c, parameters $\overline{c'}$ and the target element c'', is prefixed by the respective domain that marks the context, t, $\overline{t'}$, and t''.

A module m implements exactly one interface i. To do so, it can rely on one or more interfaces $\overline{j}$. This time, mapping signatures are supplemented by a list of statements. Assignment expressions can be used to setup fields of target model elements built from QVTo expressions. We have seven types of expressions in cQVTom: Querying a target object created from a source object, invoking a mapping s defined by an exported or imported interface j, checking the type of an expression, accessing a field f, instantiating a class c in domain t with constructor parameters $\overline{e}$, accessing the surrounding mapping's source context, and accessing an argument or variable x. This is a valid subset of QVTo's rich syntax. We now aim at showing how information hiding is enforced on this variety of concepts.

[1] Of course, one can defer static evaluation of information hiding to runtime.

Syntax:

T	$::= \overline{P}\ \overline{I}\ \overline{M}$	Transformation program
P	$::= \texttt{package}\ p\ \{\ \overline{P}\ \overline{C}\ \}$	Metamodel specification
C	$::= \texttt{class}\ c\ (\texttt{extends}\ c')^?\ \{\ \overline{F}\ \}$	Class declaration
F	$::= (\texttt{composes} \mid \texttt{references})\ f\ :\ c\ (\texttt{[1]} \mid \texttt{[*]})\texttt{;}$	Feature declaration
I	$::= \texttt{transformation interface}\ i\ (\overline{(\texttt{in} \mid \texttt{out})\ t : p\,[\overset{,}{\overline{p'}}\ \overset{,}{\overline{c}}]}^{,})\ \{\ \overline{S}\ \}$	Module interface declaration
S	$::= \texttt{mapping}\ c\texttt{@}t\ \texttt{::}\ s\,(\overline{\texttt{in}\ c'\texttt{@}t'}^{,})\ :\ c''\texttt{@}t''\texttt{;}$	Method signature declaration
M	$::= \texttt{transformation module}\ m\ \texttt{export}\ i\ (\texttt{import}\ \overset{,}{\overline{j}})^?\ \{\ \overline{O}\ \}$	Module implementation definition
O	$::= \texttt{mapping}\ c\texttt{@}t\ \texttt{::}\ s\,(\overline{\texttt{in}\ x : c'\texttt{@}t'}^{,})\ :\ c''\texttt{@}t''\ \{\ \overline{B}\ \}$	Mapping implementation definition
B	$::= \texttt{result.}f\ \texttt{:=}\ E\texttt{;}$	Assignment
E	$::= E\texttt{.late resolveone}(c\texttt{@}t)$	Trace resolution call
	$\mid E\texttt{->map}\ j\texttt{::}s\ (\overset{,}{\overline{x}})$	Mapping invocation
	$\mid E\texttt{.oclIsTypeOf}(c\texttt{@}t)$	Type checking
	$\mid E\texttt{.}f$	Feature access
	$\mid \texttt{new}\ c\texttt{@}t(\overset{,}{\overline{E}})$	Class instantiation
	$\mid \texttt{self}$	Context access
	$\mid x$	Variable access

Metamodel primitives:

$$\texttt{package}\ \mathcal{G}\ \{\ \texttt{class Object}\ \{\ \}\texttt{;}\ \texttt{class Boolean}\ \{\ \}\texttt{;}\ \texttt{class String}\ \{\ \}\texttt{;}\ \}$$

Metamodel subtyping:

$$c <: c \qquad \frac{c <: c' \qquad c' <: c''}{c <: c''} \qquad \frac{\texttt{class}\ c\ \texttt{extends}\ c'\ \{\ \texttt{...}\ \}}{c <: c' \qquad c <: \texttt{Object}} \qquad \frac{\texttt{class}\ c\ \{\ \texttt{...}\ \}}{c <: \texttt{Object}}$$

Lookup of metamodel packages, classes and features:

$$\frac{\texttt{class}\ c\ (\texttt{extends}\ c')^?\ \{\ \ldots\ \}}{\mathit{classes}_C(c) = c, \mathit{classes}_C\big((c')^?\big)}$$

$$\frac{\begin{array}{c}\texttt{package}\ p\ \{\ \overline{P}\ \overline{C}\ \}\\ \overline{P} = \overline{\texttt{package}\ p'\ \{\ \ldots\ \}} \qquad \overline{C} = \overline{\texttt{class}\ c\ (\texttt{extends}\ c')^?\ \{\ \ldots\ \}}\end{array}}{\mathit{packages}_P(p) = \overline{p'} \qquad \mathit{classes}_P(p) = \overline{c}, \big[_k\ \mathit{classes}_C\big((c'_k)^?\big)\ \big]}$$

$$\frac{\texttt{class}\ c\ (\texttt{extends}\ c')^?\ \{\ \overline{(\texttt{composes} \mid \texttt{references})\ f\ :\ c\ (\texttt{[1]} \mid \texttt{[*]})\texttt{;}}\ \}}{\mathit{features}_C(c) = \mathit{features}_C\big((c')^?\big), \overline{f} : \overline{c}}$$

Lookup of declared and implemented mapping types:

$$\frac{\begin{array}{c}\texttt{transformation interface}\ i\ \ldots\ \{\ \overline{S}\ \}\\ S = \texttt{mapping}\ c\texttt{@}t\ \texttt{::}\ s\,(\overline{\texttt{in}\ c'\texttt{@}t'})\ :\ c''\texttt{@}t''\end{array}}{\mathit{mappings}_I(i) = \bigcup_{S \in \overline{S}} \Big\{(i, s) \mapsto \big((c\texttt{@}t, \overline{c'\texttt{@}t'}) \mapsto c''\texttt{@}t''\big)\Big\}}$$

$$\frac{\begin{array}{c}\texttt{transformation module}\ m\ \ldots\ \{\ \overline{O}\ \}\\ O = \texttt{mapping}\ c\texttt{@}t\ \texttt{::}\ s\,(\overline{\texttt{in}\ x : c'\texttt{@}t'})\ :\ c''\texttt{@}t''\ \{\ \ldots\ \}\end{array}}{\mathit{mappings}_M(m) = \bigcup_{O \in \overline{O}} \Big\{(\texttt{this}, s) \mapsto \big((c\texttt{@}t, \overline{c'\texttt{@}t'}) \mapsto c''\texttt{@}t''\big)\Big\}}$$

Figure 5: cQVTom's syntax, subtyping rules, and auxiliary functions.

Module structure is well-formed:

$$\frac{\vdash \overline{I}\ \mathtt{WF} \qquad \vdash \overline{M}\ \mathtt{WF}}{\vdash T\ \mathtt{WF}} \tag{WF-Program}$$

$$\frac{\begin{array}{c}\forall k : \overline{p'}_k \subset \mathit{packages_P}^{+}(p_k) \wedge \mathit{classes_C}^{*}\big(\overline{c}_k\big) \subset \mathit{classes_P}^{*}\big(\mathit{packages_P}^{+}(p_k)\big) \\ \Delta\big[_k\ \big(a_k,\ t_k\big) \mapsto \mathit{classes_C}^{*}\big(\overline{c}_k\big) \cup \mathit{classes_P}^{*}\big(\overline{p'}_k\big) \cup \mathit{classes_P}^{*}\big(\mathit{packages_P}^{+}(\overline{p'}_k)\big) \cup \mathit{classes_P}(\mathcal{G})\ \big] \vdash \overline{S}\ \mathtt{WF}\end{array}}{\vdash \mathtt{transformation\ interface}\ i\ \mathtt{(}\overline{a = (\mathtt{in} \mid \mathtt{out})\ t : p\,\mathtt{[}\overline{p'}\ \overline{c}\mathtt{]}}\mathtt{)}\ \mathtt{\{}\ \overline{S}\ \mathtt{\}}\ \mathtt{WF}} \tag{WF-Interface}$$

$$\frac{\begin{array}{c}\mathtt{transformation\ module}\ m\ \mathtt{export}\ i\ \ldots\ \mathtt{\{}\ldots\mathtt{\}} \\ \big(\mathit{mappings_M}(m)\big)(\mathtt{this}, s) = (c_0 \mathtt{@} t_0, \overline{c'_0 \mathtt{@} t'_0}) \mapsto c''_0 \mathtt{@} t''_0 \\ \Delta(\mathtt{in}, t) \ni c \qquad \Delta(\mathtt{in}, \overline{t'}) \ni \overline{c'} \qquad \Delta(\mathtt{out}, t'') \ni c'' \\ c_0 \mathtt{@} t_0 <: c \mathtt{@} t \qquad \overline{c' \mathtt{@} t'} <: \overline{c'_0 \mathtt{@} t'_0} \qquad c'' \mathtt{@} t'' <: c''_0 \mathtt{@} t''_0\end{array}}{\vdash \mathtt{mapping}\ c\mathtt{@}t\ \mathtt{::}\ s\ \mathtt{(}\overline{\mathtt{in}\ c'\mathtt{@}t'}\mathtt{)}\ \mathtt{:}\ c''\mathtt{@}t''\ \mathtt{WF}} \tag{WF-MappingDecl}$$

$$\frac{\begin{array}{c}\mathtt{transformation\ module}\ m'\ \mathtt{export}\ i\ \ldots\ \mathtt{\{}\ldots\mathtt{\}} \Rightarrow m' = m \\ \mathtt{transformation\ interface}\ i\ \mathtt{(}\overline{a = (\mathtt{in} \mid \mathtt{out})\ t : p\,\mathtt{[}\overline{p'}\ \overline{c}\mathtt{]}}\mathtt{)}\ \mathtt{\{}\ \overline{S}\ \mathtt{\}} \\ \Delta\big[_k\ \big(a_k,\ t_k\big) \mapsto \mathit{classes_C}^{*}\big(\overline{c}_k\big) \cup \mathit{classes_P}^{*}(\overline{p'}_k) \cup \mathit{classes_P}^{*}\big(\mathit{packages}^{+}(\overline{p'}_k)\big) \cup \mathit{classes_P}^{*}(\mathcal{G})\ \big], \\ \Omega\big[\ \big(\cup_k \mathit{mappings_I}(\overline{j}_k)\big) \cup \mathit{mappings_M}(m)\ \big] \vdash \overline{O}\ \mathtt{WF}\end{array}}{\vdash \mathtt{transformation\ module}\ m\ \mathtt{export}\ i\ (\mathtt{import}\ \overline{j})^{?}\ \mathtt{\{}\ \overline{O}\ \mathtt{\}}\ \mathtt{WF}} \tag{WF-Module}$$

$$\frac{\begin{array}{c}\Delta(\mathtt{in}, t) \ni c \qquad \Delta(\mathtt{in}, \overline{t'}) \ni \overline{c'} \qquad \Delta(\mathtt{out}, t'') \ni c'' \\ \Gamma[\mathtt{self} \mapsto c\mathtt{@}t, \overline{x} \mapsto \overline{c'\mathtt{@}t'}, \mathtt{result} \mapsto c''\mathtt{@}t''], \Delta, \Omega \vdash \overline{B}\ \mathtt{WF}\end{array}}{\Delta, \Omega \vdash \mathtt{mapping}\ c\mathtt{@}t\ \mathtt{::}\ s\ \mathtt{(}\overline{\mathtt{in}\ x : c'\mathtt{@}t'}\mathtt{)}\ \mathtt{:}\ c''\mathtt{@}t''\ \mathtt{\{}\ \overline{B}\ \mathtt{\}}\ \mathtt{WF}} \tag{WF-MappingImpl}$$

$$\frac{\begin{array}{c}\Gamma(\mathtt{result}) = c\mathtt{@}t \qquad c_0 <: c' \qquad \Delta(\mathtt{out}, t) \ni c \\ \mathit{features_C}(c) \ni f : c' \qquad \Gamma, \Delta, \Omega \vdash e_0 : c_0\mathtt{@}t_0\end{array}}{\Gamma, \Delta, \Omega \vdash \mathtt{result}\mathtt{.}f\ \mathtt{:=}\ e_0\mathtt{;}\ \mathtt{WF}} \tag{WF-Assignment}$$

Expression typing and conformance checks:

$$\frac{\Gamma, \Delta, \Omega \vdash e_0 : c_0\mathtt{@}t_0}{\Gamma, \Delta, \Omega \vdash e_0\mathtt{.late\ resolveone(}c\mathtt{@}t\mathtt{)}\ \mathtt{:}\ c\mathtt{@}t} \tag{T-TraceRes}$$

$$\frac{\begin{array}{c}c_0 <: c \qquad \overline{c} <: \overline{c'} \\ \Omega(i, s) = (c\mathtt{@}t, \overline{c'\mathtt{@}t'}) \mapsto c''\mathtt{@}t'' \\ \Gamma, \Delta, \Omega \vdash e_0 : c_0\mathtt{@}t_0 \qquad \Gamma, \Delta, \Omega \vdash \overline{e} : \overline{c}\end{array}}{\Gamma, \Delta, \Omega \vdash e_0\mathtt{\text{->}map}\ i\mathtt{::}s\ \mathtt{(}\overline{e}\mathtt{)}\ \mathtt{:}\ c''\mathtt{@}t''} \tag{T-MappingInv}$$

$$\frac{\Gamma, \Delta, \Omega \vdash e_0 : c_0\mathtt{@}t_0 \qquad \big(\Delta(\mathtt{in}, t) \cup \Delta(\mathtt{out}, t)\big) \ni c}{\Gamma, \Delta, \Omega \vdash e_0\mathtt{.oclIsTypeOf(}c\mathtt{@}t\mathtt{)} : \mathtt{Boolean@}t_{\mathcal{G}}} \tag{T-TypeCheck}$$

$$\frac{\begin{array}{c}\mathit{features_C}(c_0) = \overline{f} : \overline{c} \\ \Gamma, \Delta, \Omega \vdash e_0 : c_0\mathtt{@}t_0 \qquad \big(\Delta(\mathtt{in}, t_0) \cup \Delta(\mathtt{out}, t_0)\big) \ni c_0\end{array}}{\Gamma, \Delta, \Omega \vdash e_0\mathtt{.}f_i : c_i\mathtt{@}t_0} \tag{T-Feature}$$

$$\frac{\Gamma, \Delta, \Omega \vdash \overline{e} : \overline{c'} \qquad \Delta(\mathtt{out}, t) \ni c}{\Gamma, \Delta, \Omega \vdash \mathtt{new}\ c\mathtt{@}t(\overline{e}) : c\mathtt{@}t} \tag{T-ClassInst}$$

$$\frac{\Gamma(\mathtt{self}) = c\mathtt{@}t \qquad \big(\Delta(\mathtt{in}, t) \cup \Delta(\mathtt{out}, t)\big) \ni c}{\Gamma, \Delta \vdash \mathtt{self} : c\mathtt{@}t} \tag{T-Context}$$

$$\frac{\Gamma(x) = c\mathtt{@}t \qquad \big(\Delta(\mathtt{in}, t) \cup \Delta(\mathtt{out}, t)\big) \ni c}{\Gamma, \Delta \vdash x : \Gamma(x)} \tag{T-Variable}$$

Figure 6: cQVTom's typing rules.

For any metamodel defined, a package $\mathcal{G}$ introduces the primitive data types `Object`, `Boolean`, and `String`. Respective fields have been omitted for simplicity.

Like in FJ, a subtyping relationship between classes is established by an operator <: that is based on the `extends` keyword. Subtyping is reflexive, transitive, but also antisymmetric, i.e. no cycles are permitted. For convenience, any class except `Object` inherits from `Object` by default.

We introduce auxiliary methods for metamodel and mapping lookup. These methods are utilized by the typing rules hereinafter, they are defined in Figure 5 in the lower two sections. Function $\mathit{classes}_C$ maps a class to a list of inherited classes including itself. In case that $(\texttt{extends}\ c')^?$ is omitted, c' evaluates to `Object`, and $\mathit{classes}_C(\texttt{Object}) = \epsilon$. For a given package, functions $\mathit{packages}_P$ and $\mathit{classes}_P$ compute all packages and classes directly contained in the package, respectively. And finally, for a given class, function $\mathit{features}_C$ retrieves directly contained features. Note that here—and similarly in the rest of this paper—, for brevity, we abbreviate typing judgments on sequences, writing $\overline{f} : \overline{k}$ as shorthand for $f_1 : k_1, \ldots, f_n : k_n$ (cf. [10]). Function $\mathit{mappings}_I$ creates for a given interface identifier a function that relates pairs of interface and mapping identifiers to the mapping's signature type. Analogously, $\mathit{mappings}_M$ creates such a function for any mapping defined in a module implementation – here, we use `this` for identifying the interface whose implementation is currently being defined. For any of these functions being special kinds of binary relations, the $+$ operator denotes their transitive closure. The $*$ operator is short for a functional closure on sets, for instance, $\mathit{classes}_P{}^*(P) := \bigcup_{p \in P} \mathit{classes}_P(p)$.

4.2 Typing

We build a type system in the style of the classical Hindley-Milner type system. Several ideas and many notational elements are borrowed from FJ [10]. Primary judgment of our type system is that of type well-formedness with respect to the modular structure, $\vdash T$ `WF`. To attain this goal, we must judge about the typing of expressions to determine any explicit and implicit type references. We use a type system where typing relations take the form $\Gamma, \Delta, \Omega \vdash e : t$. This reads: "In a scoped type environment Γ, Ω, Δ of variables, methods, and model elements, the term e has type t".

We capture scoping information in a type environment that consists of three parts: a variable environment Γ, a method environment Ω, and a model element environment Δ. The variable environment is a function mapping identifiers in scope to types, $\Gamma ::= \emptyset \mid \Gamma, [_{k=0}^{n}\ v_k \mapsto c_k @ t_k\ ]$. The method environment is more complex, it maps a pair of interface identifier and mapping identifier to a mapping's signature, $\Omega ::= \emptyset \mid \Omega, [_{k=0}^{n}\ (i, s_k) \mapsto \big((c_k @ t_k, \overline{c'}_k @ \overline{t'}_k) \mapsto c''_k @ t''_k\big)]$. The model element environment is a function that captures accessible model elements, $\Delta ::= \emptyset \mid \Delta, [_{k=0}^{n}\ (a_k, t_k) \mapsto C_k]$, where a_k is the access type, `in` or `out`, t_k is the model domain identifier, the pair of both mapping to C_k, the list of class identifiers that are accessible as inquired.

Notation $\Gamma[x_0 \mapsto c_0 @ t_0, \ldots, x_n \mapsto c_n @ t_n]$ is the type environment Γ updated at $x_k, k = 0..n$ to map x_k to $c_k @ t_k$. For an overlined syntax expression $\overline{x : c@t}$ type variables are represented as sequences $(x_k)_{k=0}^n$, $(t_k)_{k=0}^n$, and $(c_k)_{k=0}^n$. Then, $\Gamma[\overline{x} : \overline{c@t}]$, $\Gamma[_k\ x_k \mapsto c_k @ t_k]$, and $\Gamma[\{x_0 \mapsto c_0 @ t_0, \ldots, x_n \mapsto c_n @ t_n\}]$ are short forms for the notation mentioned above.

Type inference rules are displayed in Figure 6. They are completely syntax directed, thus defining small-step semantics. As we already mentioned, our type system is designed to prove that a modular transformation program in cQVTom is well-formed regarding the information hiding principle. A transformation program T is only then well-formed if its interface definitions and module implementations are well-formed (WF-PROGRAM).

An interface signature defines a sequence of modeling domains on packages $\overline{p}$, and for each domain a list of packages $\overline{p'}$ and classes $\overline{c}$ on which access is opened up. These elements must be contained in the respective domain's root package p (WF-INTERFACE). An environment Δ is built that maps domain names to the list of accessible classes. Inside an interface definition, mapping signatures are declared. Any of these declared signatures must be implemented by a module m with compatible types, and any type used must be accessible (WF-MAPPINGDECL). Type conformance is checked according to the Liskov principle, and accessibility is checked based on the Δ environment.

There must be exactly one implementation per interface. A module inherits model visibilities from the interface it implements, so Δ is equally configured (WF-MODULE). The Ω environment is filled with methods provided by imported interfaces plus those defined locally. A mapping implementation must have any of its signature's type accessible. Two variables plus their respective types are added to its scope, `self` and `result` (WF-MAPPINGIMPL). In the body of a mapping, the target object's features can be initialized. It must be a valid feature of the object's type, both sides of the assignment must have matching types, and the result's type must be write accessible. (WF-ASSIGNMENT).

Expression typing is obvious, insofar that we infer for each syntactical element related types, and check that the element is visible and excels a valid accessibility mode. Trace resolution and mapping invocation (T-TRACERES, T-MAPPINGINV) do not require access rights as they delegate type access to external modules. Even so must we ensure that parameter types are compatible. Global type classes are accessed via a global domain identifier $t_{\mathcal{G}} : \mathcal{G}$, rule T-TYPECHECK gives an example of use. Access to a feature demands read access only to the parent type (T-FEATURE). If an object is created, we check if its type is write accessible (T-CLASSINST). Context and variables are checked if they are in scope and their type is read accessible (T-CONTEXT, T-VARIABLE).

4.3 Properties

Soundness of the semantics with respect to a type system generally means that "well-typed programs cannot go wrong". Since we only focus on soundness of modular concepts added by us, soundness means that well-typed programs at runtime do not hurt any of our four principles. In the following, we formalize our four rules, and provide a proof sketch for each of them.

R1: Segregation of interface and implementation. Modules that implement the same interface can be exchanged while maintaining type conformance.

THEOREM 4.1. *For any pair of transformations T and T', where*

$$T = \overline{P}\ \overline{I}\ M_0 \ldots M_i \ldots M_n$$
$$T' = \overline{P}\ \overline{I}\ M_0 \ldots M'_i \ldots M_n\ ,$$

with module implementations M_i, M'_i of the same name $m = m'$, both exporting the same interface $i = i'$, and both being well-formed, i.e., $\vdash M$ `WF` and $\vdash M'$ `WF`, we can say that $\vdash T$ `WF` $\Leftrightarrow \vdash T'$ `WF`.

PROOF. In the scope of an implementation definition, methods and model types only can (and must be) dereferenced by an interface name, i.e. j `::` s and $c@t$, at the syntactical level. No assumptions concerning the actual implementation are made. Required and provided method signatures must be compatible in terms of Liskov's substitution principle, as encoded by rules WF-MAPPINGDECL for module implementations and T-MAPPINGINV for method invocations. Therefore, any module implementation remains independent of any other module implementations. □

R2: Conformance of interface and implementation. A program is only then well-formed if there exists exactly one implementation per interface. If an interface misses an implementation potential method calls cannot be resolved. If an interface is implemented multiple times it is not clearly expressed which implementation to choose resulting in nondeterministic behavior.

THEOREM 4.2. *For any interface $i \in I$, there exists exactly one implementation $m \in M$ for i. For this implementation we can find exactly one bijective mapping between implementation and interface methods $f_{m,i} : O|_m \to S|_i$, so that each method $o \in O|_m$ maps to a method $s \in S|_i$ with equal name and an equal number of arguments, $o = s$ and $|\overline{c'_o}| = |\overline{c'_s}|$, and signature types are pairwise compatible according to Liskov (contravariant argument types, covariant return types).*

PROOF. Suppose for an implementation $m \in M$ of interface $i \in I$ exists another implementation $m' \in M$ of that same interface. Then, implementations m, m' must be the same, as guaranteed by rule WF-MODULE: $m = m'$. In addition, WF-MAPPINGDECL guarantees that for any method implementation (which there must be exactly one, as we have just shown), type conformance constraints according to Liskov are met. □

R3: Method access control. In an implementation, only model types or mappings are referenced that are imported by the prefixed interface, and the interface is imported by the implemented interface.

THEOREM 4.3. *For any implementation of a module m,*

$$\texttt{transformation}\,\texttt{module}\; m\; \texttt{export}\; i\; (\texttt{import}\; \dot{\overline{j}})^?\ ,$$

if a method implementation $o \in O|_m$ references a method j' `::` o that is not locally defined, $j' \neq$ `this`, then $j' \in \overline{j}$, and the signature of o is compatible (regarding to Liskov's substitution principle) to the signature of o specified in the interface j'.

PROOF. Only mapping invocation expressions may refer to methods. There are two cases, either a called mapping o is defined locally (dereferenced using `this::`o), or the mapping is dereferenced by an interface with notation i`::`o. By inquiring the Ω environment, rule T-MAPPINGINV ascertains that only methods in scope (i.e., local or imported ones) are referenced. The same rule tests for type conformance, as well. □

R4: Model access control. In an implementation, only model types are referenced for read or write access in a specific domain if the respective access mode is declared for this model type and domain type in the interface implemented by the implementation.

THEOREM 4.4. *For any expression's inferred type, $\vdash e : c@t$, model type c must be defined as read-accessible in the respective domain t by the surrounding module's interface, except if access is delegated to another module. It must be write-accessible if features are created or modified. Additionally, for any parameter being part of a mapping or OCL operation's signature, its type $c@t$ must be defined as accessible with the correct mode (read- or write-accessible for context and input parameters, write-accessible for output and return parameter types) in the respective domain t by the surrounding module's interface.*

PROOF. For any expression that is defined in the syntax, a type is inferred by an expression typing rule. There, we can find a precondition in the form of $c \in \Delta(\texttt{in}, t) \cup \Delta(\texttt{out}, t)$ for read access checks, and $c \in \Delta(\texttt{out}, t)$ for write access checks, depending on the underlying dynamic semantics; Exceptions are T-TRACERES and T-MAPPINGINV which delegate to external modules. The same is true for method parameters (rules WF-MAPPINGDECL, WF-MAPPINGIMPL) and assignments (rule WF-ASSIGNMENT). □

Type Soundness. A program is considered as being well-typed if and only if it does not hurt the information hiding principle.

COROLLARY 4.5. *Let T be a transformation program in valid cQVTom syntax. If $\vdash T$ `WF`, transformation T does not hurt the principle of information hiding as described by rules R1 to R4.*

PROOF. From the proofs of theorems 4.1 to 4.4 immediately follows soundness of our module concept. □

Decidability. Because type inference rules are syntax directed, there is only one conclusion for each syntactic form. Evaluation will only get stuck if one of two kinds of premises remains unfulfilled, type conformance or accessibility. If and only if our type system terminates on a program with $\vdash T$ `WF`, it is well-formed. Hence the type system is decidable, and an efficient implementation exists.

5. Implementation in Xtend

For validation purposes, we prototypically implemented a transformation language *Xtend2m*[2] that includes our module concept. Xtend2m augments the *Xtend* language[3] for model-to-model (M2M) and model-to-text (M2T) transformations on EMF-based Ecore models. EMF maps Ecore metamodels to Java types. Xtend is a statically typed language that compiles to ordinary Java code. It features template expressions for M2T and cached methods for M2M, and because it is built with the Xtext framework, it comes with full-featured Eclipse editors and can be easily extended and customized. Extensibility was the primary reason we decided to use the Xtend language for a prototypical implementation of our concepts.

We exploit the fact that Xtend programs are 100% compatible with Java's type system: We utilize Java interfaces as module interfaces and Java classes as module implementations. Mapping operations are Java methods inside a class. As a consequence, Java's type checker automatically ensures that a module conforms to its interface, and enforces that only mappings marked as public are accessed from outside.

However, there are four weaknesses. First, cached methods only take care that, for a certain parameter set, the previously created element is returned instead of a new one. Second, model access restrictions can not be declared for an interface, and implementations are not statically checked for violations against restrictions. Third, module implementations must be kept independent from each other. This issue is already tackled by standard dependency injection APIs, but it is not checked if the imported interface is actually a transformation interface. And fourth, Xtend does not prescribe how a transformation's entry point must look like.

To mark classes and interfaces as transformation concepts, and to include access declarations and mapping methods with QVTo-like tracing, we designed six dedicated Java annotations. Based on these annotations, we were able to make use of an Xtend feature called *Active Annotations*. This mechanism gives language developers the chance to intercept static code analysis and transpilation to Java for two purposes. On the one hand, we can perform static type checking, and in cases of any semantic issues we can create appropriate compiler warnings and errors. These issues are then displayed at the corresponding location in the Eclipse editor. On the other hand, we can manipulate transpilation, for example, we are able to inject code into methods with a certain annotation.

Figure 7 again shows the Activity2Process transformation from the introduction, but this time it is implemented in Xtend2m rather than QVTom. All the annotations used there are going to be explained in the following paragraphs.

[2] Sources are available at `qvt.github.io/xtend2m`.

[3] Xtend is hosted at `xtend-lang.org`

```
@TransformationInterface
@ModelIn(#["activitymodel.Activity",
          "activitymodel.Action"])
@ModelOut(#["processmodel.Process"])
interface IActivity2Process extends MainMethod {
  def Process mapActivity2Process(Activity self)
}
@TransformationModule
class Activity2Process implements IActivity2Process {
  @Import extension IAction2Step

  @Creates(typeof(Process))
  override Process mapActivity2Process(Activity self) {
    result.steps = self.actions.map[mapAction2Step]
  }

  override main(List<List<EObject>> input) {
    val activity = input.head.
      filter(typeof(Activity)).head
    mapActivity2Process(activity)
    doLateResolution
  }
}
```

(a) First module

```
@TransformationInterface
@ModelIn(#["activitymodel.StartAction",
          "activityModel.StopAction"])
@ModelOut(#["processModel.Step"])
interface IAction2StepModule {
  def Step mapAction2Step(Action self)
}
@TransformationModule
class Action2Step implements IAction2Step {
  @Creates(typeof(Step))
  override Step mapAction2Step(Action self) {
     result.name = self.name
     self.succ.lateResolveOne [ result.next = it ]
     result.isStart = self instanceof StartAction
     result.isStop = self instanceof StopAction
  }
}
```

(b) Second module

Figure 7: Activity2Process example in Xtend2m

Interfaces must be indicated with `@TransformationInterface`, and classes with `@TransformationModule`. Control dependencies can be declared via `@Import`, and are mapped by the transpiler to an ordinary `@Inject`. At the same time, transformation modules are automatically injected with a factory for model creation, a module configuration class and a tracing API. Type checking makes sure that an interface implemented or imported by a transformation module is in any case annotated as a transformation interface. A dedicated interface `IMain` constitutes the entry point. A transformation is only valid if this interface is implemented by exactly one module.

We replaced cached methods with our own concept. Methods annotated with `@Creates(typeof(T))` automatically create an instance of T that is registered at our tracing API. Later on, trace resolution can be conducted in the style of QVTo, for example by calling `lateResolveOne`. In contrast to QVTo, late resolution must be triggered by an explicit call to `doLateResolution`. Any referenced model types from inside a method are checked if they are declared as accessible by the interface the surrounding module implements.

Access control can be declared for module interfaces via two annotations, `@ModelIn` and `@ModelOut`. These are parameterized by a list of model element classes. All classes in a package can be declared using a wildcard operator, `myPackage.*`.

At this time, the dependency injection framework has not been informed about available implementations. Xtend programs are typ-

```
module Activity2ProcessTransformation
Workflow {
  // load metamodels ActivityModel.ecore, Process.ecore
  // load ActivityModel instance into slot "inputModel"
  :
  component = xtend2m.mwe.ModuleLoader {
    input  = "inputModel"
    output = {
      package = "processmodel"
      slot    = "outputModel" }
    transformationModule = "Activity2Process"
    transformationModule = "Action2Step"
  }
  // persist ProcessModel from slot "outputModel"
  :
}
```

Figure 8: Activity2Process example – MWE2 workflow definition

ically orchestrated from a workflow script written for the *Model Workflow Engine* (MWE2). We built a customized workflow component that initiates the wiring and then executes the transformation. So that this can happen, module implementations must be registered. Concerning the introductory *Activity2Process* example, a workflow script must register implementations for two interfaces, `IMain` and `IAction2Step` (Figure 8).

As we have shown, transformations written in Xtend2m share all modular concepts of QVTom and key QVTo concepts. Because Xtend already comes with template expressions built-in, not only M2M, but also M2T transformations can be written. One difference concerning our module concept is that no metamodel represents the target, hence access restrictions cannot be declared.

6. Validating our Approach

To validate our approach, we chose a transformation that is practically used in a larger research project on software architecture simulation, the Palladio approach[4]. For this transformation, we are able to show that maintenance effort is significantly smaller if a transformation is structured based on our module concept.

The *Palladio* approach [1] enables the prediction of extra-functional properties at the design-time of component-based software. By analyzing simulation results, performance, scalability and reliability problems can be detected at an early stage in the development process. Component-based software architectures and typical usage scenarios are first modeled in the *Palladio Component Model* (PCM). Instances of this model are then translated to simulation code that is based on the *SimuCom* simulation framework. Other targets exist as well, for instance mappings to Plain Old Java Objects (POJO), to Enterprise Java Beans (EJB3), and to a performance prototype (ProtoCom).

Technically, the program for translating architectural models to simulation code had been implemented as an M2T transformation written in Xpand and Xtend1, both being predecessors of Xtend2[5]. M2T transformations are special cases of M2M transformations, where the target model are textual artifacts. Xpand and Xtend2 are both template-based languages, meaning that transformation logic is embedded into static text with the help of meta-tags.

We examined two maintenance scenarios that appeared recently during development. The first scenario deals with the process of refactoring the modular structure of the transformation. The second scenario is about adapting the transformation for a new requirement. We will demonstrate that the effort involved in identifying bad smells and locating concerns can be dramatically reduced with a proper modular structure and descriptive module interfaces.

[4] For details on Palladio, see `palladio-simulator.com`.

[5] As we discuss the dated dialect *Xtend1*, we refer to Xtend as *Xtend2*.

Table 1: SimuCom transformation – Data dependencies per module

Xtend Module	reliability.*	resourceenv.*	system.*	seff.*	usagemodel.*	repository.*	**LOC**
M_1: Allocation							7
M_2: Build			✗			✗	157
M_3: Calculators				✗	✗		32
$\vdots$							
M_{10}: Dummies			✗			✗	89
M_{11}: JavaCore				✗		✗	244
M_{12}: JavaNamesExt	✗		✗		✗	✗	278
M_{13}: PCMExt			✗	✗	✗	✗	480
M_{14}: ProvidedPorts						✗	260
M_{15}: Repository						✗	120
M_{16}: Resources				✗		✗	27
M_{17}: SEFFBody	✗			✗		✗	220
$\vdots$							
M_{23}: SimAllocations		✗	✗			✗	111
M_{24}: SimCalculators				✗	✗	✗	86
M_{25}: SimCalls				✗		✗	286
$\vdots$							
M_{32}: SimResources		✗					252
M_{33}: SimSEFFBody	✗			✗		✗	252
M_{34}: SimSensors							35
M_{35}: SimUsage			✗		✗	✗	253
$\vdots$							
M_{39}: SimUsageFactory			✗		✗	✗	91
LOC (Σ)							4987
Modules (Σ)	4	2	11	13	9	30	
LOC (%)	18%	7%	35%	42%	28%	87%	
Modules (%)	10%	5%	28%	33%	23%	77%	

6.1 Scenario 1: Refactoring the modular structure

One of the more recent development tasks in the Palladio project was to migrate the meanwhile deprecated Xpand templates to the Xtend2 language. Transformation templates were already modularized using the template method pattern, with the result that variants share common parts, and concretize by implementing abstract methods. Yet, former modularization did not use Java interfaces to declare which public methods implementations must provide, and in Java, it is not possible to restrict access to model elements.

Next, we utilized the Xtend2m add-on to declare proper interfaces. Results of a precursive analysis of data dependencies are depicted in Table 1 in the form of a dependence matrix. The table lists for select modules which of the six PCM packages it references. The PCM is packetized by six modeling aspects, a structural view (repository.*), a behavioral view (seff.*), an assembly view (system.*), a usage view (usagemodel.*), a resource view (resourceenvironment.*), and a view on reliability annotations (reliability.*). For instance, PCM's reliability concepts are handled—and thus referenced—by four modules: M_{12}, M_{17}, M_{28}, and M_{33} (M_{28} has been omitted in this view). These four modules share 18% of the overall 4987 lines of code (LOC).

While modules $M_1,\ldots,M_{22}$ act as generic templates for various targets including SimuCom, modules $M_{23},\ldots,M_{39}$ refine these to produce SimuCom target code. For example, M_{23} extends M_1 in order to implement several abstract methods.

According to the table, most packages reference model elements from only few packages. This indicates a low coupling regarding data dependencies. Particularly JavaNamesExt and PCMExt (M_{12} and M_{13}) exhibit a high degree of coupling. Both modules depend on four PCM packages and are called by most other modules. Inspecting their code quickly reveals that the two modules had been used to collect helper methods. This design decision issues from Xpand's inability to mix template expressions with utility functions. Both modules used to be implemented in Xtend1. However, with the advent of Xtend2, template expressions and functions are mixable. Because almost all methods are only required by single modules, they can be moved to the respective module without breaking the code.

In this example, we identified a bad smell just from studying dependencies declared in module interfaces. Thus we were able to reduce the coupling and increase cohesion between modules, making the overall transformation better understandable and maintainable. Without descriptive interfaces, we would have to reverse-engineer data dependencies manually, with the risk of missing some dependencies. On the other hand, our type interference system statically analyzes implementations against declared interfaces and identifies violations automatically.

6.2 Scenario 2: Locating concerns

In the Palladio model, software components can realize component interfaces. Interfaces in turn can extend other interfaces. When a component realizes such a chain of interfaces, it must provide operations for any interface along that inheritance chain. Until recently[6], our transformations were not aware of inheritance chains. A first step to correct the transformations is to locate places in the code where interfaces are handled. In the PCM, three manifestations of interfaces exist, all being descendants of class `Interface`, namely `OperationInterface`, `InfrastructureInterface`, and `EventGroup`. All four model elements are part of the structural view, and therefore belong to the repository.* namespace. Without having data dependencies declared, we must investigate the full code to track down relevant places. Since the SimuCom transformation employs our module concept, we can narrow down possible locations of concern by just studying module descriptions.

By looking at a transformation's module interfaces, we can tell if a module is actually authorized to access these interface concepts. With dependencies declared at the package-level, we would have to check modules with access to the repository namespace, being 30 out of 39 modules (see Table 1). Since we already have the transformation's model dependencies declared at the class-level, we can narrow down the number of modules we need to consider even further. Table 2 displays for relevant modules if they access any `Interface`-related class residing in the repository namespace. There are only 14 modules whose implementation must be examined further, reducing the amount of code to 39%. In the end, we had to edit four among these to get our task done.

Without a modular structure based on a descriptive module concept, developers need to fall back to a text-based search. However, a word-based search for "interface" leads to many false positives, because semantics are ignored. With our proposed blackbox module concept, maintenance of model transformations takes significantly less effort than with existing module concepts that do not account for data and control dependencies at the interface-level.

7. Related Work

Early formal treatment of modular concepts as they appear in general-purpose programming languages had been carried out by Burstall and Lampson [3]. More recently, modularity has been discovered as beneficial for domain-specific languages as well, for example Kang and Ryu introduced modularity to the JavaScript language [11].

The initial European workshop on composition of model transformations in 2006 marked major interest in the topic for the first time. Since then, compositionality of model transformations has been under steady research, albeit most compositional approaches

[6] `sdqbuild.ipd.kit.edu/jira/browse/PALLADIO-165`

Table 2: SimuCom transformation – Change impact analysis

Xtend Module	<: repository.Interface	Had to modify?
M_1: Allocation		
M_2: Build	✗	
⋮		
M_6: ComposedStructure	✗	
M_7: ContextPattern	✗	
M_8: DataTypes		
M_9: DelegatorClass	✗	✗
M_{10}: Dummies	✗	
M_{11}: JavaCore	✗	✗
M_{12}: JavaNamesExt		
M_{13}: PCMExt		
M_{14}: ProvidedPorts	✗	✗
M_{15}: Repository	✗	✗
⋮		
M_{19}: Sensors	✗	
M_{20}: System		
M_{21}: Usage	✗	
M_{22}: UserActions		
⋮		
M_{27}: SimContextPattern		
M_{28}: SimDummies	✗	
M_{29}: SimJavaCore	✗	
M_{30}: SimProvidedPorts	✗	
M_{31}: SimRepository	✗	
M_{32}: SimResources		
⋮		
Modules (Σ)	14	4
LOC (%)	39%	14%
Modules (%)	36%	10%

Table 3: Comparison of concepts for internal composition

Concept	QVTom	Xtend2	Kermeta	QVTr	QVTo	ATL	ETL	VIATRA2
Modules	✓	✓ class	✓ class	✓ library	✓ library	✓ module	✓ *files*	✓ namespace
Import mechanisms	✓	✓ extends	✓ inherits	✓ import	✓ access, extend	✓ uses	✓ import	✓ import
Rule inheritance	–	–	–	(✓) *unimpl.*	✓ inherits	✓ extends	✓ extends	–
Rule merging	–	–	–	–	✓ merges	–	–	–
Superimposition	–	✓ override	✓ *implicit*	(✓) *implicit*	✓ extends	✓ *implicit*	✓ *implicit*	✓ *model+ data*
Qualified namespace	✓	✓ package	✓ package	(✓) *models*	(✓) *models*	(✓) *models*	(✓) *models*	(✓) *models?*
Explicit interfaces	✓	✓ interface	✓ abstract class	–	–	–	–	–
Traces	✓	(✓) *only local*	–	–	–	–	–	–
Methods	✓	✓ def	✓ *implicit*	–	–	–	–	–
Model elements	✓	(✓) import	–	–	–	–	–	–
Information hiding	✓	✓ private	–	–	–	–	–	–

focus on reusability. In Belaunde's article on QVTo's compositional abilities [2], the author distinguishes between coarse-grained and fine-grained techniques, also known as internal and external composition. The former work on transformations and whole models, whereas the latter work at the level of mappings and model elements.

7.1 Reuse

Olsen et al. investigate possible ways to improve reusability of transformations [15], compositional techniques being among them. A more up-to-date survey and far more detailed classification of reuse techniques is given by Wimmer et al. [23, 24]. They observe that module mechanisms should support definition of access rights and restricted inheritance options. None of the presented fine-grained compositional mechanisms seems to possess blackbox characteristics. We believe the main reason is that techniques which aim at better reuse rely on invasive whitebox mechanisms, whereas we concentrate on improving maintainability. In fact, we deliberately decide in favor of maintainability: Because our approach introduces static dependencies to model elements, we even hinder reuse over metamodels, yet we can improve evolvability, understandability and type-safety.

7.2 Internal composition

Original work on modularity had been carried out in the 1990s in the field of Graph Rewriting Systems, surveyed by Heckel et al. [7]. Hiding of rewrite rules seems to be possible in all of the discussed approaches, but hiding of typed graph structures remains unsupported. More recently, Klar et al. [12] transferred MOF's package management to manage rules in MOFLON, a Triple Graph Grammar dialect. They have reuse in mind, and although rules can be hidden from imports there is no explicit interface concept.

Stratego/XT supports "meta-model extensibility through generator extensibility" [9], also known as horizontal modularity. Our approach still requires the respective modules to be modified when the models change, yet our descriptive interface helps to locate the affected modules with less effort.

Cuadrado and Molina added a rule organization mechanism called *phasing* [4] to RubyTL, a DSL embedded into Ruby. Phasing is a whitebox technique to promote modularity and internal transformation composition. Common code can be factored out, as one phase may refine rules of another phase. A phase has a scope (a pivot point, i. e. an element in the source metamodel from which a rule evolves), a precondition, by-value parameters, and a scheduling script for ordering sub-phases and binding parameters. However, their concept does not include interface descriptions to make data and rule dependencies between phases explicit.

Table 3 compares typical modularity features between languages we perceived to be most interesting and our proposed derivate, QVTom. A checkmark indicates full support, partial support if bracketed, and either the respective keyword or possible limitations are stated below. All of the observed languages support modularity to some extent.

Most concentrate on whitebox techniques for reuse matters, for example inheritance, merging, and superimposition of rules. Superimposition had been first introduced by Wagelaar to ATL and QVT [22], a technique to overlay sets of rule definitions on top of each other. QVTo is said to have an OO heritage [2] and thus only supports inheritance and superimposition.

When it comes to interface concepts, only few languages provide concepts to make traces, methods, or model elements explicit. Only internal DSLs are able to hide implementation details by exploiting concepts of high-level languages, for instance Xtend and RubyTL. Rules in Kermeta are defined as class methods in UML, so inheritance, interfaces and other UML concepts can be exploited. To our knowledge, QVTom and the Xtend2m prototype are the only approaches that introduce blackbox modularity.

7.3 External composition

The key characteristic of any external composition mechanism is that only whole transformations operating on complete models can be chained, whereas we aim at supporting finer-grained compositionality. Both *QVT and ATL* already bring along integrated blackbox composition mechanisms for orchestrating complete transformations. There is no interface concept for language constructs.

Several approaches are concerned with compositionality at the transformation level, also known as blackbox composition, mostly aiming at reusability of transformations. The most notable ones are UniTI [21] integrated into Eclipse AM3 as a GMM4CT plug-in, Wires* and TraCo. All of them follow the data-driven programming paradigm, component instances are executed as soon as models are present at all of the available input ports. None of them offers a language concept for binding model concepts, which has only

recently been proposed by Cuadrado et al. [5]. While UniTI supports a shared tracing model, it is not possible to explicitly share single traces based on concepts.

TraCo is a transformation composition framework that showcases safe composition through contractual interfaces [8]. TraCo's interfaces only provide for full models, it has neither built-in support for model access policies, nor does it include mapping operations. A TraCo component is purely data-driven, it cannot refer to external mappings or externally generated traces. A notable field of application is to ensure that only valid transformation variants can be built from available components. With our approach, type-safe configuration of variants could be performed similarly at design-time. Because our binding is resolved before runtime, there is no runtime validation intended.

8. Conclusions and Outlook

In this paper, we have introduced a novel module concept that is specially tailored for model transformations. The concept makes data and control dependencies between modules explicit, it provides interface descriptions that can hide implementation details from module users. Implementations are statically checked if they actually meet contractual obligations defined by provided interfaces. We formalized the underlying type system, and, as a proof-of-concept, integrated this approach into the Xtend language. In a case study on a real-world M2T transformation, we have shown that our module system is able to effectively reduce the effort of locating concerns involved in typical evolution scenarios.

In the near future, we plan to carry out additional case studies on M2M transformations. We are currently working on an integration of our concept into the QVTo language. Modularizing existing transformations could be assisted by an automatic clustering algorithm, provided that we can find metrics to assess cohesion and coupling of mappings. Finally, several features of common module systems remain yet unsupported. For example, only implementations can define import dependencies, modules cannot form a hierarchy. Additionally, data dependencies could be augmented with syntactic sugar, e.g. a postponed plus operator could automatically include subclasses of a named class, following a similar feature in Kermeta. Also, behavioral contracts in the spirit of Meyer's Design by Contract could complement our concept, as already proposed by Vallecillo et al. [19] for monolithic transformations.

Acknowledgments

This research has been funded by the German Research Foundation (DFG) under grant No. RE 1674/5-1.

References

[1] S. Becker, H. Koziolek, and R. Reussner. The Palladio Component Model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, Jan. 2009.

[2] M. Belaunde. Transformation composition in QVT. In *Proc. 1st Europ. Workshop on Composition of Model Transformations (CMT'06)*, TR-CTI, pages 39–46. Centre for Telematics and Information Technology, Univ. of Twente, June 2006. URL `doc.utwente.nl/66171/`.

[3] R. M. Burstall and B. W. Lampson. A kernel language for abstract data types and modules. In *Proc. Int'l Symp. on Semantics of Data Types*, volume 173 of *LNCS*, pages 1–50. Springer, 1984.

[4] J. S. Cuadrado and J. G. Molina. Modularization of model transformations through a phasing mechanism. *Software and System Modeling*, 8 (3):325–345, 2009.

[5] J. S. Cuadrado, E. Guerra, and J. de Lara. Generic model transformations: *Write once, reuse everywhere*. In *Proc. 4th Int'l Conf. Theory and Practice of Model Transformations (ICMT'11)*, volume 6707 of *LNCS*, pages 62–77. Springer, 2011.

[6] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646, 2006.

[7] R. Heckel, G. Engels, H. Ehrig, and G. Taentzer. Classification and comparison of module concepts for graph transformation systems. In *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 2: Applications, Languages, and Tools*, chapter 1. World Scientific, Oct. 1999. ISBN 9810240201.

[8] F. Heidenreich, J. Kopcsek, and U. Aßmann. Safe composition of transformations. *Journal of Object Technology*, 10:7: 1–20, 2011.

[9] Z. Hemel, L. C. L. Kats, D. M. Groenewegen, and E. Visser. Code generation by model transformation: A case study in transformation modularity. *Software and System Modeling*, 9(3):375–402, 2010.

[10] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23 (3):396–450, 2001.

[11] S. Kang and S. Ryu. Formal specification of a JavaScript module system. *SIGPLAN Not.*, 47(10):621–638, Oct. 2012. ISSN 0362-1340.

[12] F. Klar, A. Königs, and A. Schürr. Model transformation in the large. In *Proc. 6th Joint Meeting of the Eur. Software Engineering Conf. and the ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering (ESEC/SIGSOFT FSE'07)*, pages 285–294. ACM, 2007.

[13] A. MacCormack, J. Rusnak, and C. Baldwin. The impact of component modularity on design evolution: Evidence from the software industry. *Harvard Business School Technology & Operations Mgt. Unit Research Paper*, No. 08-038, 2007.

[14] Object Management Group. MOF 2.0 Query/View/Transformation, version 1.1. URL `www.omg.org/spec/QVT/1.1/`, Jan. 2011.

[15] G. K. Olsen, J. Aagedal, and J. Oldevik. Aspects of reusable model transformations. In *Proc. 1st Europ. Workshop on Composition of Model Transformations (CMT'06)*, TR-CTI, pages 21–26. Centre for Telematics and Information Technology, Univ. of Twente, June 2006. URL `doc.utwente.nl/66171/`.

[16] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[17] A. Rentschler, Q. Noorshams, L. Happe, and R. Reussner. Interactive visual analytics for efficient maintenance of model transformations. In *Proc. 6th Int'l Conf. on Theory and Practice of Model Transformations (ICMT'13)*, volume 7909 of *LNCS*, pages 141–157. Springer, 2013.

[18] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. The structure and value of modularity in software design. *SIGSOFT Softw. Eng. Notes*, 26(5):99–108, Sept. 2001. ISSN 0163-5948.

[19] A. Vallecillo, M. Gogolla, L. Burgueño, M. Wimmer, and L. Hamann. Formal specification and testing of model transformations. In *12th Int'l School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM'12)*, volume 7320 of *LNCS*, pages 399–437. Springer, 2012.

[20] M. van Amstel and M. G. J. van den Brand. Model transformation analysis: Staying ahead of the maintenance nightmare. In *Proc. 4th Int'l Conf. on Theory and Practice of Model Transformations (ICMT'11)*, volume 6707 of *LNCS*, pages 108–122. Springer, 2011.

[21] B. Vanhooff, D. Ayed, S. V. Baelen, W. Joosen, and Y. Berbers. UniTI: A unified transformation infrastructure. In *Proc. 10th Int'l Conf. on Model Driven Engineering Languages and Systems (MODELS'07)*, volume 4735 of *LNCS*, pages 31–45. Springer, 2007.

[22] D. Wagelaar. Composition techniques for rule-based model transformation languages. In *Proc. 1st Int'l Conf. on Theory and Practice of Model Transformation (ICMT'08)*, volume 5063 of *LNCS*, pages 152–167. Springer, 2008.

[23] M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, and W. Schwinger. Fact or fiction - reuse in rule-based model-to-model transformation languages. In *Proc. 5th Int'l Conf. Theory and Practice of Model Transformations (ICMT'12)*, volume 7307 of *LNCS*, pages 280–295. Springer, 2012.

[24] M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, W. Schwinger, D. S. Kolovos, R. F. Paige, M. Lauder, A. Schürr, and D. Wagelaar. Surveying rule inheritance in model-to-model transformation languages. *Journal of Obj. Techn.*, 11(2):3: 1–46, 2012.