Towards Using Constructive Type Theory for Verifiable Modular Transformations

Steffen Zschaler
King's College London,
Department of Informatics,
London, UK
szschaler@acm.org

Iman Poernomo
King's College London,
Department of Informatics,
London, UK
iman.poernomo@kcl.ac.uk

Jeffrey Terrell
King's College London,
Department of Informatics,
London, UK
jeffrey.terrell@kcl.ac.uk

ABSTRACT

Model transformations have been studied for some time, typically using a semantics based on graph transformations. This has been very successful in defining, optimising and executing model transformations, but has been less useful for providing a firm semantic basis for modular, reusable transformations. We propose a novel rendering of transformation semantics in terms of constructive type theory and show how this can be employed for expressing dependencies and guarantees of transformation modules in a formal framework.

Categories and Subject Descriptors

D2.3 [**Software Engineering**]: Coding Tools and Techniques

Keywords

Proofs-as-model-transformations, type theory, formal model driven engineering

1. INTRODUCTION

Model-Driven Engineering (MDE) focuses on using models as the central artefact of software development, and model transformations to turn them into executable code. Model transformations can encode design rules, platform choices, or even coding conventions. MDE can result in better software quality because it encourages developers to focus on high-level, domain-centered concepts, which ensures consistency of implementation and reliability of analysis.

As MDE is being used increasingly within science and industry, and transformations of interest are becoming more complex, the trustworthiness of transformations is quite rightly receiving more attention. The informality of MDE as it currently stands makes it untrustworthy and therefore potentially dangerous. If model transformations are incorrect, the MDE process can result in software of a lower quality than that produced by traditional software development. A small number of errors in a complex transformation can easily lead to an exponential number of errors in the resulting code, which may be difficult to trace and debug.

Previous work by Terrell and Poernomo [14] has attempted to solve this problem within a formal method known as Constructive Type Theory (CTT). CTT possesses a property known as the Curry-Howard Isomorphism, where data, functions and their correctness proofs can be treated as ontologically equivalent, and where a similar equivalence holds for the related trinity of typing information, program specifications and programs. A practical implication of the isomorphism is that, by proving the logical validity of a model transformation specification, we can automatically synthesize an implementation of the transformation that satisfies the specification. Following [13], we call this the *proofs-as-model-transformations* paradigm.

As transformations become more complex, there is an increasing need to be able to modularise them. Some work on this has already been done [1, 3, 4, 6–8, 11, 12, 17, 19, 20]. However, it is still difficult to safely reuse transformation modules as there are currently no techniques for expressing or verifying a transformation module's dependencies. In [14], Terrell and Poernomo showed how proofs-as-model-transformations allows us to develop a structured approach to provably correct model transformations, defining maps between class hierarchies. In this paper, we sketch how this approach can be extended to formally express contracts for transformation modules. In particular, we show how the higher-order nature of CTT can enable a natural characterisation of a transformation module's dependencies.

2. SPECIFICATION AND DEVELOPMENT OF MODEL TRANSFORMATIONS

Constructive Type Theory (CTT) as a formal method is like a conventional functional programming type system that has been extended to include logical specifications, so that a valid type inference is also a proof of program certification. For example, just as we can find terms 2 and $+$ that satisfy

$$2 : int \quad \text{or} \quad + : int * int \to int$$

as valid type inferences in a typical functional programming language, we can also develop a term t in CTT such that

$$t : \forall x : int.\exists y : int.GreaterPrimeNumber(x, y) \qquad (1)$$

Any such term t is simultaneously

- a program that, given an input x, will output a prime number y greater than x satisfying $GreaterPrimeNumber(x, y)$;
- a proof that the program meets its specification.

That is, t is proof-carrying code, a program and a certification of the program's correctness with respect to its specification (1).

We have applied the same principle to model transformations, extending the type system to accommodate EMOF like metamodels as types so that we can develop certified model transformations t by developing a type inference of the form

$$t : \forall x : Source.Pre(x) \to \exists y : Target.Post(x, y)$$

Any such term t is simultaneously

- a model transformation that, given an input model x of metamodel $Source$, will output a model y of metamodel $Target$;

- a proof that the model transformation meets its specification.

In the next section, we present a brief summary of how CTT can be used to specify and develop model transformations.

2.1 Constructive Type Theory

We sketch our version of constructive type theory (a sugared version of Coquand and Huet's Extended Calculus of Constructions, the type theory at the heart of the Coq theorem prover).

The CTT is a lambda calculus whose core set of terms, P, are given over a set of variables, V:

$$P ::= V|\lambda\ V.\ P|(P\ P)|\langle P, P\rangle|\mathsf{fst}(P)|\mathsf{snd}(P)|\mathsf{inl}(P)|\mathsf{inr}(P)|$$
$$\mathsf{match}\ P\ \mathsf{with}\ \mathsf{inl}(V) \Rightarrow P \mid \mathsf{inr}(V) \Rightarrow P$$

The lambda calculus is a functional programming language. We can compile terms and run them as programs. As such, the calculus is equipped with an evaluation semantics. Lambda abstraction and application are standard and widely used in functional programming languages such as *SML*. The term $\lambda\ x.\ P$ defines a function that takes x as input and will output $P[a/x]$ when applied to a via an application $(\lambda\ x.\ P)a$. The calculus also includes pairs $\langle a, b\rangle$, where $\mathsf{fst}(\langle a, b\rangle)$ will evaluate to the first projection a (similarly for the second projection). Case matching provides a form of conditional, so that $\mathsf{match}\ z\ \mathsf{with}\ \mathsf{inl}(x) \Rightarrow P \mid \mathsf{inr}(y) \Rightarrow Q$ will evaluate to $P[x/a]$ if z is $\mathsf{inl}(a)$ and to $Q[y/a]$ if z is $\mathsf{inr}(a)$. Evaluation is assumed to be *lazy* – that is, the operational semantics is applied to the outermost terms first, working inwards until a neutral term is reached. We write $a \rhd b$ if a evaluates to b according to this semantics.

Like most modern programming languages, the CTT calculus is typed, allowing us to specify, for example, the input and output types of lambda terms. We write

$$f : T$$

to signify that a term f has type T.

The terms of our lambda calculus are associated with the following kinds of types: basic types, e.g. integers, functional types $(A \to B)$, product types $(A * B)$, disjoint unions $(A|B)$, dependent product types $(\Pi x : t.a)$ and dependent sum types $(\Sigma x : t.b)$, where in both cases x is taken from V. The first three types have the standard meaning found in typical functional programming languages. For example,

$$t : (A \to B)$$

means that t is a function that can accept as input any value of type A to produce a value of type B.

The next two types require some explanation. A dependent product type expresses the dependence of a function's output types on its input term arguments. For example, if

$$(\lambda x.t) : \Pi x : T.F(x)$$

then the function $(\lambda x.t)$ can input any value a of type T, producing an output value $t[a/x]$ of type $F(a)$. Thus, the final output type of the function is parameterized by the input *value*.

Similarly, the dependent sum type expresses a dependence between the type of a pair's second element and the value of its first element. For example, if we have a pair

$$(\langle a, b\rangle) : \Sigma x : T.F(x)$$

then the type of b is $F(a)$.

Typing rules provide a formal system for determining what the types of lambda terms should be.

We have extended the standard way of encoding objects and classes, using record types of the same form as found in functional programming languages such as *SML*. Bidirectional and cyclic dependencies pose a technical problem to CTT. We solve this by using *co-inductive* record types. Co-induction over record types essentially allows us to expand as many references to other records as we require, simulating navigation through a metamodel's cyclic reference structure. The formal treatment of these concepts is given in [14].

2.2 Proofs as Model Transformations

The Curry-Howard isomorphism shows that constructive logic is naturally embedded within our type theory, where proofs correspond to terms, formulae to types, logical rules to typing rules, and proof normalization to term simplification. Consider the set of well-formed formulae WFF, built from exactly the same predicates that occur in our type theory. We can define an injection asType from WFF to types of the lambda calculus as in Fig. 1.

A	$\mathsf{asType}(A)$
$Q(x)$, where Q is a predicate	$Q(x)$
$\forall x : T.P$	$\prod x : T.\mathsf{asType}(P)$
$\exists x : T.P$	$\Sigma x : T.\mathsf{asType}(P)$
$P \wedge Q$	$\mathsf{asType}(P) * \mathsf{asType}(Q)$
$P \vee Q$	$\mathsf{asType}(P)\|\mathsf{asType}(Q)$
$P \Rightarrow Q$	$\mathsf{asType}(P) \to \mathsf{asType}(Q)$
$\perp$	$\perp$

Figure 1: Definition of asType, an injection from WFF to types of the lambda calculus.

The isomorphism tells us that logical statements and proofs correspond to types and terms. We assume we have a proof inference system for constructive logic $\vdash_{Int}$ (similar to the inference systems taught in undergraduate logic classes, where $\Gamma \vdash_{Int} P$ means that a proposition P can be logically deduced from a set of assumptions Γ).

Theorem 1 Let $\Gamma = \{G_1, \ldots, G_n\}$ be a set of premises. Let $\Gamma' = \{x_1 : G_1, \ldots, x_n : G_n\}$ be a corresponding set of typed variables. Let A be a well-formed formula. Then the following is true. Given a proof in constructive logic of $\Gamma \vdash_{\mathsf{Int}} A$ we can use the typing rules to construct a well-typed proof-term p : *asType*(A) whose free proof-term variables are Γ'. Symmetrically, given a well-typed proof-term $p : \mathsf{asType}(A)$ whose free term variables are Γ', we can construct a proof in constructive logic $\Gamma \vdash A$. □

Because the isomorphism holds, *we will often omit the use of* asType *and use logical connectives and quantifiers instead of their computational counterparts (and vice versa) where there is ambiguity (for example, we will write* $\forall$ *instead of* Π *if the context makes it clear that a dependent product is being employed).*

The key implication of this theorem is that

- types can be considered to be specifications of functional programs and
- an inhabitant of a specification type can be considered to be both a program that satisfies the specification and a proof of this satisfaction.

These results are entailed by the following.

Theorem 2 Let $\Gamma = \{G_1, \ldots, G_n\}$ be a set of premises. Let $\Gamma' = \{x_1 : G_1, \ldots, x_n : G_n\}$ be a corresponding set of typed variables. Let $\forall x : T.\exists y : U.P(x, y)$ be a well-formed $\forall\exists$ formula.

If

$$\vdash p : \mathsf{asType}(\forall x : T.\exists y : U.P(x, y))$$

is a well typed term, then

$$\vdash \forall x : T.P(x, \mathsf{fst}(px))$$

is provable. □

The theorem means that, given a proof of a formula $\forall x : T.\exists y : U.P(x, y)$, we can automatically extract a function f that, given input $x : T$, will produce an output fx that satisfies the constraint $P(x, fx)$.

Our notion of proofs-as-model-transformations essentially follows from this theorem. Given that we have the machinery to type the structure of arbitrary metamodels, a model transformation between two metamodels $Source$ and $Target$ can be thought of as a functional program

$$t : Source \to Target$$

Such a program can be specified as set of constraints over instances of the input $x : Source$ and output $y : Target$ metamodels. In the simplest case, we can consider these constraints to be of a precondition $Pre(x)$ that is assumed to hold over input metamodel instances $x : Source$, and a postcondition relationship $Post(x, y)$ that holds between x and required output metamodel instances $y : Target$.

Given types $Source$ and $Target$ to represent the source and target metamodels, and constraints as logical formulae over terms of the metamodels, we can then *specify* the transformation by a formula

$$\forall x : Source.Pre(x) \to \exists y : Target.Post(x, y)$$

After that, we can attempt to find a certified transformation by identifying an inhabiting term t of

$$t : \forall x : Source.Pre(x) \to \exists y : Target.Post(x, y)$$

If we look at the meaning of the types (recall that $\forall$ corresponds to a dependent product Π and $\exists$ to a dependent sum Σ), we see that t must be a function that takes in any input x of type $Source$ and returns a pair

$$tx = \langle w, p \rangle$$

In order to synthesize a provably correct model transformation, we apply the extraction mapping over t according to Theorem 2: this will give us the required model transformation $\mathsf{fst}(t) = w$ and a certification of the transformation's correctness, a proof $\mathsf{snd}(t) = p$.

3. MODULARISING MODEL TRANSFORMATIONS

There has been considerable research interest in modularising model transformations for some time already. The approaches proposed and studied so far, may be characterised by the granularity of modules that they provide: At a first level, we can distinguish *internal* composition of transformation rules from *external* composition of entire model transformations [10]. We can further differentiate internal composition into *inter-rule* composition, where entire rules are taken to be modules, and *intra-rule* composition, where rules themselves can be composed of finer-grained modules. In the following, we will briefly discuss each of these compositions in turn.

3.1 External Composition

External composition takes entire model transformations to be modules that can be independently reused and composed. Early research on external composition focused mainly on languages and tools for describing and executing such compositions of reusable model transformations. This has led to early work on MDA components [4], megamodelling [5], transformation chaining [3, 6, 17], and transformation configuration [19].

As all of this work considers transformations as black-box components to be composed into larger components, the 'signature' or 'interface' of a transformation becomes important. These terms refer to the information that can be obtained about a transformation without inspecting its implementation. Initial work on external composition [12, 18] defined transformation signatures by two sets of metamodels: one typing the models that the transformation consumed and another typing the models produced by the transformation. Later research [6] found that this is not always sufficient information for safely composing transformations. In particular, endogenous transformations transform between models of the same metamodel, but may well only address particular elements within this metamodel. Information about the metamodel thus becomes useless when composing a set of endogenous transformations. In addition, some endogenous transformations may be intended to be used with a fixpoint semantics (invoking them until no more changes occur), which makes composing them even more complex. It was concluded in [6] that in addition to the metamodel, there needs to be information about the particular subset of model elements that are used or affected by a transformation. In parallel to this work, [17] also identified a need to include information about the technical space of models (e.g., MOF or XML) into the transformation signature. Alternatively, some of this information has been encapsulated by wrapping models as components themselves, providing interfaces for accessing and manipulating the model in a fashion independent of the technical representation [12].

3.2 Internal Composition

Internal composition considers modules of a finer granularity than entire model transformations. *Inter-rule composition* considers individual rules to be modules, while *intra-rule composition* considers even finer-grained modules, i.e. parts of rules.

3.2.1 Inter-rule Composition

A number of transformation languages consider transformation rules to be the unit of modularity. A number of mechanisms are provided for composing rules into transformations, including implicit and explicit rule invocation, and rule inheritance [3, 11]. Approaches inspired from graph transformation—for example, VMT [16]—even allow for chaining of individual transformation rules. Module superimposition [20] applies the notion of superimposition from feature-oriented software development [2] to the development of transformation modules, allowing individual rules to be overridden by rules from superimposed modules.

All of these techniques create some flexibility in allowing developers to exchange or independently evolve rules. However, they do not distinguish a rule's interface from its implementation, which means that rules and rule compositions cannot be verified or understood modularly without inspecting the complete implementation of each rule. Furthermore, some evaluations have shown that there are scenarios where the modularisation capabilities available at the level of complete rules are not sufficient [7, 8, 11].

3.2.2 Intra-rule Composition

To improve modularisation capabilities, a number of mechanisms have been proposed that allow parts of rules to become units of modularity. Balogh and Varró [1] describe how matching and creation patterns can be defined as standalone units of modularity, and composed into more complex patterns for use in transformation rules. Johannes et al. [9] allow rules to be composed and generated from a number of pattern instantiations annotated to the source metamodel.

While these approaches clearly improve the modularity capabilities of inter-rule composition approaches, they still do not enable modular verification or understanding.

In summary, while most of these techniques provide some assurances with respect to the syntactic correctness of models produced from a composed transformation (if only by virtue of the fact that they abide by a metamodel), there is very little support for modular reasoning about semantic properties. In the next section, we propose a formal encoding of transformation semantics, which allows us to provide modular reasoning and verification about semantic transformation properties.

4. MODULAR TRANSFORMATION FUNCTIONS

Higher-order quantification is the ability to make statements that are generic or parametrised over other functions, statements or proofs of statements. The proofs-as-model-transformations idea, when combined with higher-order quantification, allows us to formally treat modularity in transformations and transformation specifications.

The higher order nature of CTT allows us to parametrize statements over variables that stand as placeholders for other statements. This is achieved by introducing a higher-order universe type $Prop$ of all propositions: the type allows us to treat logical statements as forms of data to be quantified over, just like integers or strings. We can therefore define specifications that are parametrized with respect to arbitrary sub-requirements. For example, we can parametrize the specification of a transformation from UML to relational databases as

$$\forall \begin{array}{l} SubReq : (UML * RDBS) \\ \to Prop. \end{array} \left(\begin{array}{l} \forall x : UML. \\ \exists y : RDBS. \\ Post(x,y) \wedge \\ SubReq(x,y) \end{array} \right) \quad (2)$$

The predicate variable $SubReq$ stands for any subrequirement we might have over the input and output model instances of the transformation. It could, for example, stand for a proposition $CTS(x, y)$, which asserts that the number of tables in a relational database $y : RDBS$ is greater than or equal to the number of classes in an input UML diagram $x : UML$. Given a proof of (2), the variable $SubReq$ could then be replaced with CTS, yielding an *instantiated* version of the generic specification:

$$\forall x : UML.\exists y : RDBS.Post(x, y) \wedge CTS(x, y)$$

This instantiated formula can be considered to be a version of the generic specification, rendered specific to a particular requirement about classes and tables.

We can combine this treatment of parametrised specifications with the Curry-Howard isomorphism to define a notion of transformation modularity that includes a formal treatment of certified parameters. This is done by quantifying, not only over subrequirements, but also over arbitrary programs and proofs.

For example, we are able to parameterise a specification over an assumed input proof of a sub-requirement. Consider the modular transformation dependency given in Fig.2. The right hand module represents a generic, structural transformation between UML object diagrams such that an input object $A : UO$ is mapped to a new root object $B : UO$, standing in $A.b$ number of relations to a list of objects, $CL : List(UO)$. Assume this mapping has been defined by the predicate $Map(A, B, CL)$. The specification is generic over the properties that might hold over each $C \in CL$ (in particular, its attribute c): this subrequirement is defined as a predicate variable $SubReq(A, C)$.

We can define a type for the modular transformation with parameters representing both the subrequirement and an assumed *proof* *pr* of the subrequirement as follows

$$\forall \begin{array}{l} SubReq : UO * UO \to Prop. \\ pr : \forall A : UO.\exists C : UO.SubReq(A, C). \\ \left(\begin{array}{l} \forall A : UO.\exists B : UO.\exists CL : List(UO). \\ Map(A, B, CL) \wedge \\ \forall C \in CL.SubReq(A, C) \end{array} \right) \end{array} \quad (3)$$

The parameter *pr* stands for a proof that, given any A, we can find a C such that $SubReq(A, C)$. It would be employed in the proof of the composed, instantiated transformation to certify that a particular data transformation between the source and target can be plugged in to the structural transformation.

Note that the parameter $SubReq$ in the example above predicates over individual model elements, as opposed to Map, which deals with the entire model structure. This formally expresses the separation of concerns between the structural transformation Map and its parameter, which can only specify data transformations.

Thus, higher-order quantification over proofs and subrequirements allows us to

- formally represent modular specifications, parametrised over desirable subrequirements.
- by the Curry-Howard ismorphism, instantiation of such parametrised specifications correspond to certified modular transformations.

5. CONCLUSIONS

As model transformations become more important to software development, systematic development of these transformations becomes itself more important. We have shown our current ideas on how constructive type theory can be used to formally express the interfaces of, dependencies between, and contracts supported by transformation components. A key enabling factor in this has been the use of higher-order type theory, which allowed us to quantify (i.e., parametrise) over predicates and proofs of these predicates. This has enabled a transformation module to express precisely what it expects of other transformation modules with which it can be composed.

In the present paper, we have presented the essential idea of our approach using an extremely academic example only. We are currently working to apply this idea to examples of modular transformations from [9, 11] and hope to report on this more extensively in a further publication.

6. REFERENCES

[1] András Balogh and Dániel Varró. Pattern composition in graph transformation rules. In *1st European Workshop on Composition of Model Transformations (CMT'06)* [10], pages 33–37.

[2] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.

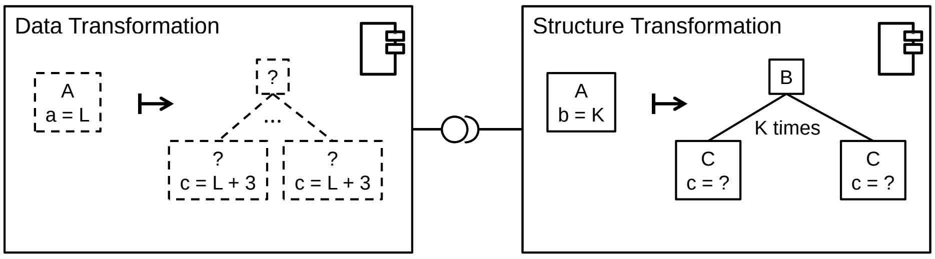

Figure 2: High-level representation of two transformation components

[3] Mariano Belaunde. Transformation compositon in QVT. In *1st European Workshop on Composition of Model Transformations (CMT'06)* [10], pages 39–45.

[4] Jean Bézivin, Sébastien Gérard, Pierre-Alain Muller, and Laurent Rioux. MDA components: Challenges and opportunities. In Andy Evans, Paul Sammut, and James S. Willans, editors, *Proc. 1st Int'l Workshop Metamodelling for MDA*, pages 23–41, York, UK, 2003.

[5] Jean Bézivin, Frédéric Jouault, Peter Rosenthal, and Patrick Valduriez. Modeling in the large and modeling in the small. In Uwe Aßmann, Mehmet Aksit, and Arend Rensink, editors, *Proc. MDAFA 2003/04*, volume 3599 of *Lecture Notes in Computer Science*, pages 33–46. Springer Berlin / Heidelberg, 2005.

[6] Raphaël Chenouard and Frédéric Jouault. Automatically discovering hidden transformation chaining constraints. In Schürr and Selic [15], pages 92–106.

[7] Thomas Cleenewerck and Ivan Kurtev. Separation of concerns in translational semantics for DSLs in model engineering. In *ACM Symposium on Applied Computing*, pages 985–992, 2007.

[8] Arda Goknil and N. Yasemin Topaloglu. Composing transformation operations based on complex source pattern definitions. In *1st European Workshop on Composition of Model Transformations (CMT'06)* [10], pages 27–32.

[9] Jendrik Johannes, Steffen Zschaler, Miguel A. Fernández, Antonio Castillo, Dimitrios S. Kolovos, and Richard F. Paige. Abstracting complex languages through transformation and composition. In Schürr and Selic [15], pages 546–550.

[10] A. G. Kleppe. 1st European workshop on composition of model transformations (CMT'06). Technical Report TR-CTIT-06-34, Centre for Telematics and Information Technology, University of Twente, June 2006.

[11] Ivan Kurtev, Klaas van den Berg, and Frédéric Jouault. Evaluation of rule-based modularization in model transformation languages illustrated with ATL. In *Proc. 21st Annual ACM Symposium on Applied Computing (SAC'06')*, pages 1202–1209, April 2006.

[12] Raphaël Marvie. A transformation composition framework for model driven engineering. Technical report, University of Lille 1, 2004. LIFL technical report 2004-n10.

[13] Iman Poernomo. Proofs-as-model-transformations. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *Proc. 1st Int'l Conf. on Theory and Practice of Model Transformations (ICMT'08)*, volume 5063 of *Lecture Notes in Computer Science*, pages 214–228. Springer, 2008.

[14] Iman Poernomo and Jeffrey Terrell. Correct-by-construction model transformations from partially ordered specifications in Coq. In Jin Song Dong and Huibiao Zhu, editors, *Proc. 12th Int'l Conf. Formal Methods and Software Engineering (ICFEM 2010)*, volume 6447 of *Lecture Notes in Computer Science*, pages 56–73. Springer, 2010.

[15] Andy Schürr and Bran Selic, editors. *Proc. Int'l Conf. on Model Driven Engineering Languages and Systems (MoDELS'09)*, volume 5795 of *LNCS*. Springer, 2009.

[16] Shane Sendall, Gilles Perrouin, Nicolas Guelfi, and Olivier Biberstein. Supporting model-to-model transformations: The VMT approach. In Arend Rensink, editor, *Proc. MDAFA 2003*, 2003. Published as CTIT Technical Report TR–CTIT–03–27, University of Twente.

[17] Bert Vanhooff, Dhouha Ayed, Stefan Van Baelen, Wouter Joosen, and Yolande Berbers. UniTI: A unified transformation infrastructure. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Proc. 10th Int'l Conf. Model Driven Engineering Languages and Systems (MoDELS'07)*, volume 4735 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2007.

[18] Andrés Vignaga, Frédéric Jouault, María Cecilia Bastarrica, and Hugo Brunelière. Typing in model management. In Richard Paige, editor, *Proc. 2nd Int'l Conf. on Theory and Practice of Model Transformations (ICMT'09)*, volume 5563 of *Lecture Notes in Computer Science*, pages 197–212. Springer-Verlag, 2009.

[19] Dennis Wagelaar and Ragnhild Van Der Straeten. A comparison of configuration techniques for model transformations. In Arend Rensink and Jos Warmer, editors, *Proc. ECMDA-FA 2006*, volume 4066 of *LNCS*, pages 331–345. Springer, 2006.

[20] Dennis Wagelaar, Ragnhild van der Straeten, and Dirk Deridder. Module superimposition: A composition technique for rule-based model transformation languages. *Software and Systems Modeling*, 9:285–309, 2010.