Demystifying Model Transformations: An Approach Based on Automated Rule Inference

Mangala Gowri Nanda  Senthil Mani  Vibha Singhal Sinha  Saurabh Sinha
IBM India Research Lab
{mgowri,sentmani,vibha.sinha,saurabhsinha}@in.ibm.com

Abstract
Model-driven development (MDD) is widely used to develop modern business applications. MDD involves creating high-level abstractions of domain concepts and successively transforming these abstractions to design-level models and, eventually, code-level artifacts. Although many tools exist that support transformer creation and verification, tools that help users in understanding and using transformers are rare. In this paper, we present an approach for assisting users in understanding model transformations and debugging their input models. Our approach uses automated program-analysis techniques to analyze the transformer code and compute information, such as constraints under which a transformation may fail or be incomplete. This information is used to compute rules that can be used to check whether an input model violates transformation constraints, and to support general user queries about a transformation.

To evaluate the approach, we conducted empirical studies using real applications. The results of the studies indicate that our approach can be effective in inferring useful rules. The main benefit of our approach is that it lets users efficiently diagnose the failure, or identify the usage conditions, of a transformation, without examining the transformation source code.

1. Introduction
Model-driven development (MDD) is a paradigm of software development that is based on the use of software modeling as a primary form of expression. It is widely used to develop modern business applications because it enables application design in terms of high-level “domain” concepts (from the problem space) instead of low-level “programming” concepts [(15), (7)]. Typically, projects that follow a MDD methodology create a series of models at various levels of abstraction—that are successively refined—before actual code is created. As an example, a business analyst outlines the application workflow as process models. These process models are used by the system analyst to design the application; the design may be captured in notations such as UML. Next, the UML models are used to generate code and test-case skeletons. Developers add application logic to the code skeletons, whereas testers fill in the testing logic in the test-case skeletons.

There are many factors that determine the effectiveness and efficiency of MDD. First, manual transformation of one model to another can be error-prone and time-consuming; therefore, automation of this step is essential. To address this problem, projects routinely use model transformations to automate the conversion of one model to another downstream artifact. Second, given a transformer that translates an input model to an output model, the transformer code must be verified to ensure that the output model is correct. To address this problem, existing research has developed many techniques for verifying model transformations (8; 14; 12; 2; 5; 11). Finally, even if a transformation is correct, the user of the transformer might have to spend significant effort in understanding the transformation; to be able to provide the correct input model to generate the desired output. Unlike the first two problems, the third problem, which requires development of techniques to assist in understanding model transformations, has mostly been overlooked. This is the problem that we address in this paper.

Illustration of the problem  Typically, a transformation is tightly coupled to the input and output model schema. Therefore, very often the transformer code assumes that the input model contains all valid elements; the code does not contain error checks (e.g., checking a reference variable for a null value) to avoid potential runtime exceptions caused by invalid input model elements. Such implicit assumptions in the code may not be obvious to the transformer user. Thus, when a transformation fails with an exception, the user faces the onerous task of manually examining the transformer code to understand which assumptions were violated, and what input model elements cause the assumptions to be violated. For
another example, a transformation may generate an output model in which an element is missing; this may or may not be caused by input constraint violations. To understand why the element is missing, the user would have to wade through the source code to identify the condition under which the element is not generated.

Ideally, users would want to consider a transformation function to be a black-box that they would rather not examine to understand the transformation logic. In many cases, the source code of a transformation may not even be available for inspection. Therefore, a technique that helps users in understanding a transformation without requiring them to examine the source code of the transformation would be useful. In current state of the practice, the transformer author usually documents the details of the transformation in plain text. Such documentation can be incomplete, incorrect; it may also become outdated as the transformer code evolves.

Our approach To address these problems and overcome the debugging limitations of existing tools, we present a new approach for assisting users in understanding the output model generated by a transformation. Our approach uses program-analysis techniques to analyze a transformer \( \tau \) and extract automatically two types of constraints that can be useful for explaining the output of \( \tau \). First, our approach recovers constraints on the input model under which an execution of \( \tau \) may fail with an exception. Using this information, the user can identify—without examining the source code of \( \tau \)—the input model elements that caused the failure. Second, our approach extracts constraints on the input model under which an output model element is either generated or has a certain value. This information is useful for explaining to the user the reason why an output model element is generated.

Without getting into any technical details, we now present an example of our analysis. For the transformation code in Figure 4, to check for nullness of type src at line 9, we generate the following predicates: \((\text{source}.\text{isSimple}() = \text{true}) \land (\text{source}.\text{getType}() = \text{null})\). Each predicate that is rooted in the source element is of interest as it implies a constraint on the input source model. In this paper we also show how these predicates are mapped to the domain model.

This raises two queries: How is this different from standard precondition analysis? What is different about its application to model-to-model transforms as against standard Java programs?

Unlike other precondition analyses where an access path is strictly composed of variables and fields, our precondition analysis is novel in the way it captures a predicate as a complete access path with functions and variables nested within. So a precondition could be as simple as \( \text{source}.\text{getType}() = \text{null} \) and as complex as: \( \text{source}.\text{getContextualArtifacts()}.\text{iterator()}.\text{next()}.\text{getName}()\text{.equals(source}.\text{getArtifacts()}.\text{iterator()}.\text{next()}.\text{getAttributes()}.\text{iterator()}.\text{next()}\text{.getType()} = \text{true} \). and still be mapped semantically to the input model. Which brings us to the second question. We ran

our constraint generator on a standard Java program, Apache ant and generated similar looking constraints. However, due to a lack of context, we could not apply the rules usefully. In the model-to-model context, these rules can be used to construct a model verifier that, given an input model for transformer \( \tau \), checks whether the model violates the constraints of \( \tau \). The verifier can be used in an interactive mode to guide the user in creating a valid input model, thus ensuring that the transformer neither fails nor generates an unexpected output model. The rules can also be used to build a querying tool, which the user can use to understand transformation semantics. For example, the user can query the tool to identify the input model elements from which an output model element is derived.

We implemented the approach and performed empirical evaluation using transformers for real applications. Our results indicate that, for the subjects considered, our approach can infer a significant number of useful rules. We also conducted a user study, in which we compared the efficiency of users in identifying and fixing problems with incorrect input models to a transformer. In the study, all the users performed the debugging tasks much faster when they were guided by the inferred rules than when they were not.

Contributions The main benefit of our approach is that it provides automated support for diagnosing the cause of a failure of a transformation and understanding the semantics of a transformation. Thus, it reduces the manual effort required to perform such tasks, which not only makes the process more efficient but also less prone to errors. Another benefit of the approach is that it is applicable in cases in which the source code of the transformer is not available for examination. This occurs frequently in practice as third-party tools are often used to perform automated model transformation. In such cases, the tool provider (by using our rule-inference approach) can add verification and querying capabilities to the tool, and thus improve the usefulness of their tool.

The contributions of the paper are:

- A static-analysis-based approach for inferring rules from model transformer code, and application of the rules for validating input models and explaining output models.
- Empirical studies conducted using different types of models and transformers that illustrate the benefits of the approach.

In this paper, we present our approach for model-to-model transformations. Our approach can be extended to include model-to-code or other model-to-text transformations too, which are also frequently used in model-driven development. We discuss briefly—and leave it to future research to investigate in detail—the application of our approach to such transformations.

The rest of the paper is organized as follows. In the next section, we formally define models and introduce an example that we use to illustrate our approach. In Section 3,
we present our approach for inferring rules and using the rules for model verification and transform comprehension. Section 4 presents empirical results and Section 5 discusses related work; finally, Section 6 summarizes the paper and lists directions for future research.

2. Definitions and Example

In this section, we provide definitions that are required to present our approach and present an example transformation that we use in the rest of the paper to illustrate our approach. First, in Section 2.1, we introduce the example transformation, INFOTRANS. Then, in Section 2.2, we formally define metamodels and models, and illustrate the input and output models for INFOTRANS. Finally, in Section 2.3, we present fragments of the transformer code for INFOTRANS and illustrate the problems that our approach addresses.

2.1 Example

To illustrate the concepts described in this paper, we use an application called INFOTRANS that takes as input a domain-specific information model, converts the model to a database schema, and creates a set of services the let users interact with the data. A domain subject-matter expert provides the input information model using a pre-defined metamodel. INFOTRANS converts the input model to a UML class model, using the model-to-model transformation framework provided by the Rational Software Architect(RSA). Next, the class model is converted, using model-to-text transformations, to create a database-schema definition file and Java classes that implement the data services. For the purpose of this paper, we focus on the model-to-model transformation part of INFOTRANS only.

INFOTRANS is a modified version of an application that was developed in context of a real project at IBM Research; the goal of the project is to build domain-specific workbenches for managing requirements in different types of projects. (Table 2) provides information about the size of INFOTRANS along with the other subjects used in our empirical studies; Figure 4 presents code fragments for three of the transformer methods.)

2.2 Metamodels and Models

A metamodel describes the structure or the abstract syntax of a model in terms of the types of elements and relations that the model may be constructed from. We define a metamodel more formally as follows.

**DEFINITION 1. (Metamodel)** A metamodel $\mathcal{M}$ is a tuple $(\mathcal{E}, \mathcal{R}, \mathcal{P}, \delta_\mathcal{R}, \delta_\mathcal{P}, e^i_\mathcal{E})$. $\mathcal{E} = \{e^1_\mathcal{E}, \ldots, e^k_\mathcal{E}\}$, $k \geq 1$, is a set of element types. $\mathcal{R} = \{r_1, \ldots, r_k\}$, $k \geq 1$, is a set of relation types. $\mathcal{P} = \{p_1, \ldots, p_k\}$, $k \geq 1$, is a set of properties. $\delta_\mathcal{R} : \mathcal{E} \rightarrow (\mathcal{R} \times \mathcal{E}, \text{String}, \text{cardinality}, \text{isSimple})$ maps an element type to its related element types, where String represents the name of the related element type; cardinality (‘one’, ‘many’, or ‘NA’) represents the number of related element types. $\delta_\mathcal{P} : \mathcal{E} \rightarrow \mathcal{P}(\mathcal{P})$ maps an element type to its associated properties. $e^i_\mathcal{E} \in \mathcal{E}$ is the unique root element type: $\exists! e^i_\mathcal{E} \in \mathcal{E} : \delta_\mathcal{P}(e^i_\mathcal{E}) = (r, e^i_\mathcal{E}), \forall r \in \mathcal{R}$.

Examples of some commonly used schema languages for defining metamodels include ECore2, XSD3, and MOF4. These languages serve the purpose of specifying the syntax of models, and thus, are analogous to language grammars that define the syntax of programming languages. Figure 1 shows the INFOTRANS input metamodel, defined using ECore. The metamodel contains seven types of elements and two types of relations: a containment relation and an inheritance relation. Three of the element types—Artifact, ContextArtifact, and EnumArtifact—represent information entities. An Artifact is a key business entity that has a lifecycle associated with it. In the generated UML model, such entities have operations associated with them for creating, deleting, updating, and retrieving all, or particular, instances of the entity. A ContextArtifact is an information entity that are available on the Artifact; operations for creating, updating, or deleting such entities are available on the Artifact which contains this entity. An EnumArtifact stores a predefined list (or, an enumeration) of values to select from. The generated output model has operations for retrieving the list of values from, adding a value to, or deleting a value from, an EnumArtifact.

An information entity is composed of Attributes, each of which has four properties associated with it: name, multiplicity, isSimpleType and type. The type property can be an element.

\[ \mathcal{E} = \{\text{DataModel, Artifact, ContextArtifact, EnumArtifact, Attribute, Annotation, ThisPackage}\} \]

\[ \mathcal{R} = \{C : \text{containsFrom}\} \]

\[ \mathcal{P} = \{\text{name, type, value, multiplicity, isSimple}\} \]

\[ \delta_\mathcal{R} : \mathcal{E} \rightarrow \{\{C, \text{Artifact, attributes, many}\}, \{C, \text{ContextArtifact, contextArtifacts, many}\}, \{C, \text{EnumArtifact, enumArtifacts, many}\}\} \]

\[ \text{Artifact} \rightarrow \{\{C, \text{Attribute, attributes, many}\}\} \]

\[ \text{ContextArtifact} \rightarrow \{\{C, \text{Attribute, attributes, many}\}\} \]

\[ \text{EnumArtifact} \rightarrow \emptyset \]

\[ \text{Attribute} \rightarrow \{\{C, \text{Annotation, annotations, many}\}\} \]

\[ \text{Annotation} \rightarrow \emptyset \]

\[ \text{ThisPackage} \rightarrow \{\{I, \text{Annotation, NA}\}\} \]

\[ \delta_\mathcal{P} : \mathcal{E} \rightarrow \{\{\text{name}\}\} \]

\[ \text{Artifact} \rightarrow \{\{\text{name}\}\} \]

\[ \text{ContextArtifact} \rightarrow \{\{\text{name}\}\} \]

\[ \text{EnumArtifact} \rightarrow \{\{\text{name}\}\} \]

\[ \text{Attribute} \rightarrow \{\{\text{name}, \text{isSimple}, \text{cardinality}\}\} \]

\[ \text{Annotation} \rightarrow \{\{\text{name}, \text{value}\}\} \]

\[ \text{ThisPackage} \rightarrow \{\{\text{name}, \text{value}\}\} \]

\[ e^i_\mathcal{E} = \text{DataModel} \]

**Figure 1.** The input ECORE metamodel for transformer INFOTRANS.
Figure 2. An input model (an instance of the metamodel shown in Figure 1) for transformer INFOTRANS.

Figure 3. Mapping of some of the input model elements to output model elements (created by the transformer for INFOTRANS).

The function $\delta_r$ maps each element type to its relationship with other element types. The result of the mapping is a 4-tuple consisting of a relation type, an element type, a name, and a cardinality. Except for the relationship between Annotation and ThisPackage, all other relationships are containment relations. Similarly, $\delta_p$ maps each element type to the properties associated with the element.

**Definition 2. (Metamodel access path)** Let $(E, R, P, \delta_r, \delta_p, e_r)$ be a metamodel. Let $\pi$ be a sequence $e_1, e_2, \ldots, e_k$, a sequence $e_1, e_2, \ldots, e_k, p$, or a sequence $e_1, e_2, \ldots, e_k, c$, $k \geq 1$, such that (1) $e_i \in E$, (2) $e_1 = e_r^i$, (3) for $k \geq 2$, $\delta_r(e_{k-1}) = (r_k, e_k, n_k, c)$, where $r_k$ is a containment relation, (4) $p = \delta_p(e_k)$, and (5) $c$ is an integer constant, such that $e_k$ has cardinality 'many'. A metamodel access path $\pi_{model}$ is formed by replacing each $e_i, 2 \leq i \leq k$, with $n_i$ in $\pi$.

Thus, a metamodel access path is a sequence of containment relations that starts at the root element type and ends at either an element type or a property type (that is a associated with the preceding element type). For example, in the input metamodel for INFOTRANS, DataModel.contextArtifacts.attributes is an access path that ends at an element type, whereas DataModel.artifacts.attributes.isSimple is an access path that ends at a property type.

Given a metamodel $\mathcal{M}$, a model can be constructed by creating instances of the element types, relations, and properties specified in $\mathcal{M}$. We define a model constructed from a metamodel as follows.

**Definition 3. (Model)** A model $M$ constructed from a metamodel $\mathcal{M} = (E, R, P, \delta_r, \delta_p, e_r)$, $E = \{e_1, \ldots, e_k\}, k \geq 1$, is a set of elements such that each $e \in E$ is an instance of an element type $e' \in E$. $P$ is a set of property instances $(p, \text{val})$, where $p \in P$ and $\text{val}$ is a value assigned to property $p$. $\delta_r : (R \times E) \rightarrow \mathcal{P}(E)$ maps a model element to its related elements. $\delta_p : E \rightarrow \mathcal{P}(P)$ maps a model element to its associated property instances. $e_r \in E$ is the unique root element of the model.
Figure 2 shows an input model—an instance of the metamodel (Figure 1)—for INFOTRANS. The example illustrates an information model for the requirements-gathering activity in a business process transformation project. The business analyst identifies the focal business processes (BusinessProcess). For each business process, a set of gaps (Gap) between the as-is and to-be process are identified. For each gap, a solution for bridging the gap (GapResolution) is identified and a priority (priority) is assigned to it. Because a GapResolution exists in context of a Gap, it is defined as a ContextArtifact type. Attribute priority stores pre-defined priority values (such as, high, medium, low); therefore, it’s type is an enumeration type.

For the INFOTRANS application, the input ECORE model is converted to a UML class model using a transformer. Instead of presenting an output UML model and metamodel, we illustrate how the INFOTRANS transformer generates output model elements from input model elements. Figure 3 shows the input-to-output mappings for some of the model elements. For example, for each Artifact in the input model, the transformer creates a Class, with a set of operations and parameters, in the output model.

2.3 Transformer

A transformer takes as input a model conforming to a given metamodel and produces as output model conforming to another metamodel or a text base artifact such as code.

**Definition 4. (Model-to-model transformer)** A model-to-model transformer \( \tau : M_I \rightarrow M_O \) is a program that given an input model \( M_I \) (an instance of metamodel \( M_I \)) generates an output model \( M_O \) (an instance of metamodel \( M_O \)).

Several tools exist that support the creation of transformers. Transformers can be written in general-purpose programming languages (e.g., Java) or scripting languages (e.g., XSLT). Some tools, such as RSA, provide specialized transformation authoring frameworks.

The INFOTRANS transformer is implemented in Java using the RSA transformation authoring framework; Figure 4 shows the code fragments for three of the methods.

The `execute1()` method is used to convert an Attribute instance in the input model to an instance of Property in the output model. If the `isSimple` property of Attribute is set to true (line 8) and the value of type property of Attribute is String (line 9), the method searches for the corresponding UML type (line 10) and sets the type for Property in output model (line 12).

Function `execute2()` is invoked during the course of transformation to create operations and associations for all elements of type Artifact. It takes as arguments the input and output model instances. It iterates over all the Artifacts and invokes `handleComplexType()` for each instance, passing the list of contained attributes and corresponding output class as arguments. The output Class instance is created by a method (not shown in the figure) that transforms each Artifact / ContextArtifact / EnumArtifact element in the input model to a Class in output model. `handleComplexType()` in turn converts all input model attribute elements, whose property `isSimple` is set to false, to associations in the output model. `handleComplexType()` iterates over each attribute (line 16), checks if a `ThisPackage` annotation is available on the `Attribute` (line 21), and then creates the association (code not shown).

The bottom part of Figure 4 lists three potential exceptions that can occur during the execution of the transformer. If, in the input model, an Attribute instance does not have a type property, `attr.getType()` returns `null` at line 7, which causes a null-pointer exception at the dereference of `type_src` at line 9. An array-index exception occurs on...
3. Our Approach

In this section, first, we present an overview of our approach (Section 3.1). Then, we discuss different components of the approach, which include inference of constraints using static analysis (Section 3.2), generation of transform rules (Section 3.3), and support for model verification models and transform comprehension (Section 3.4).

3.1 Overview

Figure 5 presents an overview of our approach, which consists of two phases: (1) rule generation, and (2) model verification and transform comprehension. The first phase generates the transformation rules in two steps. In the first step, the approach analyzes the transformer code to compute exception and output constraints. (We define these constraints more formally in Section 3.2.) The computed constraints include conditions on transformer inputs that cause the transformer to terminate with a runtime exception or generate an incomplete output model. The code analyzer also takes as an input user-provided filter specifications that it uses to remove some of the computed constraints that are uninteresting from the user’s perspective. For example, some of the constraints are filtered because the transformer-authoring framework enforces certain input conditions, which the code analyzer is unaware of; therefore, the analyzer can compute constraints that cannot be true in that specific framework, although they could be true in a different authoring framework.

The information generated during the first step is at a low level of abstraction: the constraints are stated in terms of program variables. To support verification and comprehension, these constraints need to be stated at a higher level of abstraction. Thus, the second step of Phase 1 translates the constraints to state them in terms of input and output metamodel elements. Exception constraints are translated to verification rules, whereas output constraints are converted to querying rules.

Phase 2 of our approach supports model verification and transform comprehension using the rules inferred in the first phase. The verification rules inferred in Phase 1 can be used to construct a verifier that, given an input model, identifies the rules that are satisfied by the model. The verifier can also be used in an interactive mode to guide the user in the task of creating an input model. Thus, in this mode, the user avoids creating an invalid input model to begin with, instead of first creating the model and then running it through the verifier. The querying rules can be used to help the user understand why certain output model elements are not generated in an execution of a transformer.

3.2 Constraint Inference

To compute constraints, we leverage the null-dereference analysis implemented in a tool called XYLEM (13). The goal of the XYLEM analysis is to identify a program path along which a dereference at a given statement can be null. Starting at a statement $s$, that dereferences variable $v$, XYLEM performs a backward, path-sensitive and context-sensitive analysis to identify such a path. During the analysis, it propagates a set of abstract state predicates backward in the control-flow graph (CFG). The analysis starts with a predicate asserting that $v$ is null, and updates states during the...
path traversal. If the updated state becomes inconsistent, the path is infeasible and the analysis stops traversing the path. The algorithm returns after identifying a feasible path.

For our approach, we modified and extended the XYLEM analysis in several ways: (1) We extended the predicates from the standard nullary, unary and binary predicates to include multi-variable predicates that include function calls and all the related parameters in the access path. (2) In addition to identifying null-pointer exceptions, the extended analysis identifies potential class- and (limited forms of) array-index exceptions, and (3) instead of identifying one feasible path (to a null dereference), the analysis identifies constraints on input variables along all paths.

First, we discuss the general state propagation and path exploration that is performed by the XYLEM analysis. Following that, we present our approach that leverages the path-sensitive analysis to identify exception constraints (for null-pointer, class-, and array-index exceptions) and output constraints. Finally, we discuss the filter feature which is used to ignore not-useful constraints.

### 3.2.1 Path-sensitive analysis for computing constraints on inputs

Before describing the analysis, we present definitions that formalize the constraints on input variables computed by the algorithm.

#### Abstract state and input constraints

The analysis propagates a set of abstract state predicates backward in the program.

**Definition 5. (Abstract state)** An abstract predicate $\gamma$ is a predicate of the form shown in Figure 6. An abstract state $\Gamma$ is a conjunction of abstract predicates.

Figure 6 shows the abstract predicates tracked by the analysis. These include three types of predicates on reference variables (predicates 1–3); a predicate on boolean variables (predicate 4); and two types of predicates on integer variables (predicates 5 and 6). One of the contributions of this work includes includes two additional predicate—one on types of variables (predicate 7) and another on sizes of collection objects (predicate 8); We discuss these predicates further in Section 3.2.2. Finally, one of the novelties of this work is modified rule for an access path-- the grammar rule labeled 9 in Figure 6.

**Definition 6. (Code access path)** A code access path is a sequence of dereferences $v_1.v_2.\ldots.v_k$, $k \geq 1$, where $v_i$, $1 \leq i < k$, is a reference variable or a method that returns a reference type, and $v_k$ is a method, a reference, an integer, or a boolean variable. For $v_i = m(x_1,\ldots,x_j), j \geq 0$, each $x_i$ is a code access path.

Rule 9 allows method names to appear, with arbitrary nesting, in an access path. Such method names represent calls to external methods (i.e., methods that are not part of the transformer being analyzed). An example of such an access path is
g MOCK lighten createURI("pathmap://url"),getContents(),UMLPackage.PACKAGE).getOwedElements().get(0)!=null).

**Definition 7. (Path constraint)** Let $\rho = (s_1,\ldots,s_k), k \geq 2$ be a program path (a sequence of statements) from the entry statement of the program to statement $s$. Let $\gamma$ be an abstract predicate on a variable used at $s_k$. The path constraint $C(\rho,\gamma)$ is the state $\Gamma_\rho$ at $s_1$ such that if the predicates in $\Gamma_\rho$ are true, $\gamma$ is true at $s_k$.

In many cases, a path constraint with respect to a predicate and a statement, is a sufficient condition for the predicate to be true at the statement. However, this is not true in general because of limitations of static analysis. We discuss this further in Section 3.5.

**Definition 8. (Input constraint)** Let $\textit{paths}(s)$ be the set of paths from the entry statement of a transform program to statement $s$. Let $I = \{i_1,\ldots,i_k\}, k \geq 1$, be the set of input variables to the program. Let $\gamma$ be an abstract predicate on a variable used at $s$. The input constraint $C_I(\gamma,s)$ is the disjunction $\bigvee_{\rho \in \textit{paths}(s)} \Gamma_\rho[I]$, where $\Gamma_\rho[I]$ contains the predicates in $\Gamma_\rho$ with respect to the variables in $I$.

The input constraint is a formula in the Disjunctive Normal Form (DNF), where each disjunct represents the constraint (on input variables) along one program path from the entry to the given statement $s$.

Figure 7 presents procedure ComputeConstraints, which performs the path-sensitive and flow-sensitive XYLEM analysis. ComputeConstraints is a recursive procedure that, given an input abstract predicate $\gamma$ and an input statement $s$, computes the constraints on input variables under which $\gamma$ evaluates to true at $s$.

The procedure uses a standard worklist-based approach to compute a fix-point solution over the state predicates (lines 2–16). Each element of the worklist is a pair consisting of a statement and a state. The procedure initializes the state to $\gamma$ and adds $(s,\Gamma)$ to the worklist (line 2). Then, it iteratively removes an element $(s,\Gamma)$ from the worklist processes.
procedure ComputeConstraints 
\begin{itemize}
\item input $s$ \quad statement to start backward analysis from
\item $\gamma$ \quad stating predicate at $s$
\item output $C_I$ \quad constraints on input variables
\item global $CS$ \quad call stack of methods
\item $\sigma(s, \Gamma)$ \quad summary information at a call site $s$ that maps an incoming state $\Gamma$ to an outgoing state
\end{itemize}

begin

1. initialize state $\Gamma$ to $\{\gamma\}$; initialize worklist with $(s, \Gamma)$
2. while worklist $\neq \emptyset$ do
3. remove $(s, \Gamma)$ from worklist
4. foreach predecessor $s_p$ of $s$ do
5. \quad if $s_p$ is not the entry and not a call then 
6. \quad compute $\Gamma'$ for the transformation induced by $s_p$
7. \quad if $\Gamma'$ is consistent then 
8. \quad add $(s_p, \Gamma')$ to worklist if not visited
9. \quad else if $s_p$ is a call that invokes $M$ then 
10. \quad $\Gamma' = \sigma(s_p, \Gamma)$
11. \quad if $\Gamma' = \emptyset$ then // no summary exists 
12. \quad push $\Gamma$ onto $CS$; $\Gamma' = \text{map} \Gamma$ to the exit of $M$
13. \quad ComputeConstraints(exit node of $M, \Gamma'$); pop $CS$
14. \quad $\Gamma'' = \text{map state at the entry of } M$ to $s_p$
15. \quad add $\Gamma''$ to $\sigma(s_p, \Gamma)$
16. \quad add $(s_p, \Gamma''')$ to worklist if not visited
17. \quad if $CS = \emptyset$ then // method not being analyzed in a specific context 
18. \quad if this is the entry method of $\tau$ then 
19. \quad $C_I = C_I \lor \Gamma$ // add path constraint to DNF
20. else foreach call site $s_c$ that calls this method do 
21. \quad $\Gamma'' = \text{map } \Gamma$ to $s_c$; ComputeConstraints($s_c, \Gamma''$)
end

Figure 7. The new XYLEM analysis used to compute constraints on input variables under which the given predicate $\gamma$ evaluates true at the given statement $s$.

Interprocedural path exploration Figure 8 shows the state transformations that occur at some of the statements. The notation $\Gamma[x/y]$ represents the state with each syntactic occurrence of variable $x$ replaced by $y$. Consider the state transformation at statement $x = r.f$. The updated state contains the predicates in the incoming state, with each predicate on $x$ replaced with a predicate on $r.f$, and predicate $\{r \neq \text{null}\}$. At the call to an application method, $x = r.m()$, the updated state consists of $\{r \neq \text{null}\}$ and the transformation induced on $\Gamma$ by $r.m()$. (Function $\sigma$ computes the transformation of the given state by the given method.) Our extensions to the original XYLEM analysis include modifications to some of the state transformations 5–8 (highlighted with an asterisk). Transformations 5–7 are related to the analysis of class-cast exceptions; we discuss them further in Section 3.2.2. Transformation 8 is related to the extension to rule 8 in Figure 6. At a statement $x = m()$ that calls an external method, the incoming state is updated by replacing occurrences of $x$ with the expression for the method call. This lets the analysis identify conditions involving external method calls, where the parameters of the method have dependences on input variables.

To perform efficient analysis of called methods, the procedure uses summary information at call sites. The summary information at a call site maps an incoming state $\Gamma$ to an outgoing state $\Gamma'$ to which the called method transforms $\Gamma$. Using the summary information, the procedure avoids analyzing a method multiple times for the same state. On reaching a call site, the procedure first checks whether summary information exists for the current state (lines 9–11). If no summary exists, the algorithm descends into the called methods to analyze them (lines 12–14). It uses a call stack to ensure context-sensitive processing of called methods. After returning from the called method, the analysis saves the summary information for reuse in subsequent traversals (line 15).

On reaching the entry of the method that is not being analyzed in a specific context (line 17), the algorithm ascends to all call sites that call the method (lines 20–21). If the entry of the transform is reached, the algorithm adds the state predicates as a disjunct to the input constraints (lines 18–19).

Example 1. Consider the XYLEM analysis, starting at line 9 with $\gamma = \langle \text{type}._\text{src} = \text{null} \rangle$, for the transformer code shown in Figure 4. On traversing the condition at line 8, the analysis adds predicates $\langle \text{attr}.\text{isSimple()} = \text{true} \rangle$ $\langle \text{attr} \neq \text{null} \rangle$ to the state. At this point, the state contains three predicates. Statement 7 transforms predicate $\langle \text{type}._\text{src} = \text{null} \rangle$ to $\langle \text{attr}.\text{getType()} = \text{null} \rangle$. When the analysis reaches the assignment to attr in line 1, it has found a constraint on input variable source:

\begin{itemize}
\item $(\text{source} \neq \text{null}) \land$
\item $(\text{source}.\text{isSimple()} = \text{true}) \land$
\item $(\text{source}.\text{getType()} = \text{null})$
\end{itemize}

\[\square\]

\[\text{A context-sensitive analysis propagates states along interprocedural paths that consist of valid call-return sequences only—the path contains no pair of call and return that denotes control returning from a method to a call site other than the one that invoked it.}\]
Next, we discuss the computation of exception and output constraints, which leverages procedure ComputeConstraints.

3.2.2 Computation of exception and output constraints

Figure 9 presents algorithm TransformerAnalysis that computes exception and output constraints for a given transformer. We define an exception constraint more formally as follows.

**Definition 9. (Exception constraint)** An exception constraint is a constraint \( C_{I(x)}(\gamma, s) \), where \( \gamma \) represents the condition under which a runtime exception can occur at \( s \) in some execution of the program.

For example, if statement \( s \) dereferences variable \( v \), \( \gamma = \langle v = \text{null} \rangle \), and \( C_{I(x)}(\gamma, s) \) represents the conditions on input variables under which a null-pointer exception can be thrown at \( s \). Our current approach handles Java null-pointer exceptions, class-cast exceptions, and array-index exceptions.

**Null-pointer exceptions** To compute exception constraints for potential null-pointer exceptions, TransformerAnalysis processes each statement in the transformer that dereferences a variable to check whether a null-pointer exception could occur at that statement (lines 1–4). For a dereference of variable \( v \) at statement \( s \), the algorithm initializes \( \gamma \) to \( \langle x = \text{null} \rangle \); then, it calls a procedure ComputeConstraints, which computes the conditions on input variables under which \( \gamma \) evaluates to true at \( s \).

**Class-cast exceptions** To identify class-cast exceptions, the algorithm keeps track of abstract predicates on types of reference variables. For a reference variable \( v \), predicate \( \langle \text{type}(v) \in T \rangle \) asserts that \( v \) points to an instance of one of the types in the set \( T \); the negation of this predicate, \( \langle \text{type}(x) \nsubseteq T \rangle \) asserts that the type of \( v \) is not in the set \( T \).

Similar to null-dereference analysis, the algorithm (Figure 9) starts at a statement \( s \) where a class-cast exception could be raised, and initializes the state with a predicate that asserts the condition under which the exception occurs at the statement. For a typecast statement \( x = (T)y \), \( \gamma \) is initialized to \( \langle \text{type}(y) \nsubseteq (\text{subtypes}(T) \cup \text{null}) \rangle \) (line 6), which states that \( y \) is neither null nor of a type that can be cast to \( T \). Then, the algorithm calls analyzeMethod to get back input constraints under which \( \gamma \) is true at \( s \) (lines 7–8).

State transformations 5–7 shown in Figure 8 are relevant for the analysis of class-cast exceptions. For example, transformation 7 (that occurs at a statement \( x \text{ instanceof } T \)) adds two predicates to the incoming state \( \Gamma \): the first predicate asserts that \( x \) cannot be null because, in the Java semantics, a \text{null} is not an instance of any type; the second predicate constrains the type of \( x \) to be a subtype of \( T \). Transformation 6 is similar, but it does not add \( \langle x \neq \text{null} \rangle \) to the state because, in Java, a \text{null} can be cast to any type.

Given two type predicates on a variable \( v \), they are resolved as follows:

\[
\langle \text{type}(s) \in T_1 \rangle \land \langle \text{type}(s) \in T_2 \rangle = T_1 \cap T_2, \quad \text{if } (T_1 \cap T_2) \neq \emptyset \quad \text{conflict, otherwise}
\]

**Example 2.** Consider the example program fragment shown below. To determine whether a class-cast exception can occur at line 3, the analysis initializes the state with predicates \( \langle a \neq \text{null} \rangle \) and \( \langle \text{type}(a) \in \{A2, A21, A22\} \rangle \). State 2 generates the predicates \( \langle a \neq \text{null} \rangle \) and \( \langle \text{type}(a) \in \{A2, A21, A22\} \rangle \). The resolved set of predicates now consists of \( \gamma_1 = \langle a \neq \text{null} \rangle \) and \( \gamma_2 = \langle \text{type}(a) \in \{A2, A22\} \rangle \). Then, state 1 generates predicate \( \langle \text{type}(a) \in \{A1\} \rangle \), which is inconsistent with \( \gamma_2 \). Therefore, the path (1, 2, 3) is infeasible and, consequently, no class-cast exception can occur at line 3.

```java
public class A {}
public class A1 extends A {}
public class A2 extends A {}
public class A21 extends A2 {}
public class A22 extends A2 {}
...
```

**Array-index exceptions** We perform a limited analysis of statements, such as statement 20 in the INFOTRANS code (Figure 4), that retrieve a value from a collection using an
integer constant as the index value. At such a statement \(s\): \(c.get(intConst)\), if the size of collection \(c\) is less than or equal to \(intConst\), an array-index exception is thrown. Our extension performs a simple analysis: it computes the conditions on input variables under which (1) \(s\) is reached with \(c \neq null\), and (2) there is no condition statement, on which \(s\) is directly or indirectly control dependent,\(^7\) that checks the size of \(c\).

TransformerAnalysis iterates over the statements that retrieve an element from a collection using a constant index (line 7). For each such statement \(s: c.get(intConst)\), the algorithm calls ComputeConstraints with predicates \((c \neq null)\) and \((c.size \leq intConst)\) (lines 8–9). If there is no check on \(c.size\) (line 10), the algorithm does adds the generated constraints to \(C_f(\exists)\) (line 11); otherwise, the algorithm assumes that no array-index exception can occur at \(s\) and ignores the computed constraints.

Note that although, the algorithm adds a predicate of the form \(v.size \leq intConst\) to the initial state, it does not analyze how \(v.size\) is actually updated in the program—for example, by statements that add elements to the collection. Thus, for array-index exceptions, the analysis computes constraints that assert that the \(c.get(intConst)\) statement is reached (with no check on \(c.size\)) and that the size of \(c\) is less than or equal to \(intConst\). It does not compute the actual conditions under which \(c \leq intConst\).

**Example 3.** For the call to \(get()\) at statement 20 in Figure 4, the analysis calls ComputeConstraints with the initial state containing only \(\gamma_1 = \langle attr.getAnnotations().size \leq 0 \rangle\). It does not add a null predicate on \(attr.getAnnotations().size\) to the state because a filtering rule states that \(getAnnotations()\) does not return \(null\), as discussed later in this section. At statement 18, it picks up predicate \(\gamma_2 = \langle attr.isIsSimple() \neq true \rangle\). Statement 17 updates both the predicates by replacing \(attr\) with \(attritr.next()\). Next, at statement 16, analysis does not add a predicate for the loop condition. (We explain in Sections 3.3 and 3.4 how loop predicates on \(iterator.hasNext()\) are irrelevant for rule generation and model verification.) At the entry of \(handleComplexTypes()\), the analysis ascends to the call site at line 30 of \(execute2()\). Using, the actual-to-formal parameter matching, it updates the state predicates by replacing \(attritr\) with \(artifact.-getAttributes()\). Thus, at this point \(\gamma_1\) is:

\[
\text{artifact.getAttributes().next().getAnnotations().size} \leq 0.
\]

Continuing in the same manner through statements 28, 27, 26, and 24, the analysis computes \(\gamma_1\) at entry as:

\[
\text{source.getArtifacts().next().getAttributes().}
\text{next().getAnnotations().size} \leq 0.
\]

Predicate \(\gamma_2\) is updated similarly.

Column 2 of Table 1 shows the constraints inferred for the three INFOTRANS exceptions (Figure 4). For brevity, we have replaced occurrences of “\(\cdot .next()\)” in the constraints with “\(\cdot \)” in the table.

**Output constraints** We define an output constraint formally as follows.

**Definition 10. (Output constraint)** Let \(v\) be an output variable of a transform \(\tau\). Let \(s_v\) be the exit statement of \(\tau\). Let \(\text{redefs}(s_v, v)\) be the reaching definitions of \(v\).\(^8\) An output constraint \(C_{out}(v)\) is the disjunction \(\bigvee_{d \in \text{redefs}(v)} C_l(\gamma_d, s_{cd})\), where \(d\) is control dependent on \((s_{cd}, L)\).\(^9\)

---

\(^7\)A statement \(s\) is control dependent on a predicate \((p, L)\), if there are two branches out of \(p\) such that by following the branch labeled “\(L\)”, \(s\) is definitely reached, whereas by following the other branch, \(s\) may not be reached.

\(^8\)A reaching definition defined for a statement–variable pair \((s, v)\) is a statement \(d\) such that \(d\) defines \(v\) and there exists a path from \(d\) to \(s\) in the program such that no statement along the path (other than \(d\)) defines \(v\).
Each reaching definition initializes. The algorithm examines the statement \( s_{cd} \) on which \( d \) is control dependent (line 15). It then initializes \( \gamma \) to a predicate asserting that the condition at \( s_{cd} \) evaluates such that \( d \) is reached (line 16), and, finally, calls ComputeConstraints to identify the input conditions under which \( \gamma \) evaluates to true (line 17).

**Example 4.** Consider statement 12 in the INFOTRANS code fragment (Figure 4), which defines the property type in the target object. For the output variable corresponding to property type, statement 12 is one of the reaching definitions. The analysis starts at the control dependence of this statement—statement 11—with a predicate \((\text{ptype} \neq \text{null})\). At statement 10, \( \text{ptype} \) is replaced with the external method call to `UMLUtilities.findType(...)`. Next, the analysis picks up predicate \( \langle \text{type}_\text{src}.\text{equals}('\text{String}') = \text{true} \rangle \) at statement 9 and \( \langle \text{attr}.\text{isIsSimple}() = \text{true} \rangle \) at statement 8. After processing the assignments at statements 7 and 1 (as discussed in Example 1), the analysis identifies the following conditions at the entry of the method:

\[
\text{source}.\text{isIsSimple}() = \text{true} \land \\
\text{source}.\text{getType}().\text{equals}('\text{String}') = \text{true} \land \\
\text{UMLUtilities}.\text{findType}(...) \neq \text{null}.
\]

The third conjunct is not related to an input variable and is, therefore, excluded from the constraints output by the analysis.

After computing the constraints, TransformerAnalysis performs post-processing (line 17) to remove predicates that do not involve input variables. Such predicates are of the form in which the code access path ends at an external method call, such that the actual parameters of the call are not dependent on transformer inputs (as illustrated in Example 4).

**Constraint filtering** Our approach uses filters to improve the results computed by our tool. These filters exclude certain predicates that are uninteresting from the transformer user’s perspective. We divide the filters into two categories: “invalid constraints” and “bug constraints.” The first category filters predicates that cannot be true because of constraints imposed by the transformer authoring framework. This category of filter prevents false predicates from being added to the state during the analysis. For example, if EMF were used to serialize the input model file into a Java object, any list accessed from such objects and corresponding iterators cannot be null. Thus, at a dereference \( \text{list.f} \) of such a list, predicate \((\text{list} \neq \text{null})\) need not be generated. For example, in INFOTRANS, the call to \( \text{attr}.\text{getAnnotations()} \) at line 20 returns an EMF EList, which cannot be null. Therefore, the analysis does not add predicate \((\text{attr}.\text{getAnnotations()} \neq \text{null})\) to the state at that statement (as mentioned earlier in Example 3). In our experience, the use of only a few such filters can improve the results of the analysis significantly by removing many false predicates.

The second category filters out constraints that indicate potential bugs in the transformer code. Such constraints, although relevant for the transformer author, are uninteresting from the transformer user’s perspective. In fact, in a scenario where the transformer author uses the approach to compute constraints, the author would either fix the potential bugs or filter out the constraints before computing the transform rules.

### 3.3 Rule Generation

The second step of Phase 1 of our approach (Figure 5) converts code-level constraints to model-level rules.

**Definition 11.** (Rule predicate) A rule predicate \( \psi \), defined with respect to an access path in a metamodel \( \mathcal{M} \), is a predicate of the form shown in Figure 10.

A rule predicate is defined in terms of a metamodel access path, and specifies constraints on the path. For example, for the INFOTRANS, DataModel.artifacts.attributes.type = null is a rule predicate. A transform rule is defined over a conjunction of rule predicates.

**Definition 12.** (Transform rule) A transform rule \( \Psi \), defined with respect to a transformer \( \tau : \mathcal{M}_1 \rightarrow \mathcal{M}_2 \) is a rule, of the form shown in Figure 10. The left-hand side (the antecedent) is a conjunction of rule predicates. The right-hand side (the consequent), which states the result of the rule, is either an exception or the creation of an output metamodel element or property (defined as an access path).
Table 1 shows the transform rules generated for the three exception constraints and one output constraint for INFO-TRANS. For example, the first rule states that if the isSimple property of DataModel.artifacts.attributes is true and the type property is null, a null-pointer exception occurs in the transformer.

Note that whereas a code constraint is a DNF formula (a disjunction of conjunctions) over abstract predicates, a transform rule is a conjunction of rule predicates. The rule generator translates each conjunct (or, a path constraint) in a code-level input constraint to a transform rule. Thus, an input constraint with \( n \) path constraints leads to the generation of \( n \) transform rules. We define rules as a conjunction, instead of a DNF formula, because it enables the identification and elimination of duplicate rules. For example, an exception constraint may cause null-pointer exceptions at several statements in the transformer. The code analysis will compute the same constraint for each of these statements. Consequently, the rule set will have multiple rules that have the same antecedent and consequent (null-pointer exception); such duplicate rules are removed during rule generation. Moreover, such a representation is natural because a rule provides a top-down view as opposed to a bottom-up view provided by code constraints. For an exception or an output statement, a code constraint shows different alternative input conditions that can cause the exception to be raised or the output to be generated, respectively (a bottom-up view). A transform rule, in contrast, shows different alternative results (exception or output generation) that can occur for a particular conjunction of input conditions (a top-down view).

The constraint-to-rule translation requires converting an abstract predicate \( \gamma \) to a rule predicate \( \psi \), which, in turn, essentially involves converting a code access path \( \pi_{code} \) to a metamodel access path \( \pi_{model} \). Thus, the core of the rule-generator component is the access-path translation step. Recall from Definitions 6 and 2 that \( \pi_{code} \) is a sequence of dereferences of variables or method return values, whereas \( \pi_{model} \) is a sequence of containment relations that is composed of metamodel element names and possibly ending with a property. For each reference variable \( v \) or method \( m() \) in \( \pi_{code} \), the rule generator has to identify the metamodel access path to replace \( v \) or \( m() \) with.

For example, consider the first exception constraint and its corresponding transform rule in Table 1. To perform the translation, the rule generator has to replace variable source with metamodel access path DataModel.artifacts.attributes, method isIsSimple() with property isSimple, and method getType() with property type.

To do the translation, our approach uses a mapping file; Figure 11 shows a partial XML representation of the mapping information for INFO-TRANS. A mapping file, in general, links metamodel element names and properties to entries in the transformer inputs. For INFO-TRANS, this requires linking the input ECORE metamodel element names and properties to Java methods and fields.

In the representation shown in Figure 11, each metaModelElement entry maps a metamodel element name or property type to Java method names. For example, property isSimple maps to method isIsSimple() in the input Java class. Similarly, the element type named artifacts maps to method getArtifacts(). Each methodName entry in the file states the metamodel access paths for the inputs and outputs of a “main” method in the transformer—a method that is invoked to perform a transformation. (In the RSA transformation authoring framework, a transformer can have multiple main methods; this may not be true for other frameworks.) For example, method execute1() takes as input a Java class that corresponds to the metamodel access path DataModel.artifacts.attributes or the path DataModel.contextArtifacts.attributes; its output Java class corresponds to the access path Model.classes.properties in the output UML metamodel. Note that, an input or output class can correspond to more than one access paths. Similarly, for method execute2(), the input Java class maps to access path DataModel, whereas the output class maps to access path Model.

Using this mapping information, the rule generator can perform the constraint-to-rule translation by mapping code access paths to metamodel access paths.
Example 5. Consider the constraints and transform rules shown in Table 1. For the code access path of the first exception constraint, the rule generator creates two metamodel access paths, and, therefore, two transform rules. It replaces source (the input Java class name of execute1()) with DataModel.artifacts.attributes in the first rule, and DataModel.contextArtifacts.attributes in the second rule. Next, it replaces method name isIsSimple() with property isSimple in the first rule, and getType() with property type in the second rule. Thus, the exception constraint for the null-pointer exception is translated to two transform rules.

Example 6. Consider the access path for the predicate on types in the second constraint in Table 1. For this constraint, the relevant entry method is execute2(); therefore, using the parameter-mapping information for execute2(), the rule generator translates source to DataModel. Next, using the third mapping rule in Figure 11, the rule generator replaces getArtifacts.* with artifacts. In general, an access of an element in a collection (e.g., c.next()) in πcode corresponds to a metamodel element with cardinality ‘many’ in πmodel. Similarly, the generator translates getAttributes().* and getAnnotations.*. Finally, the generator replaces get(0) with 0 to compute the translated metamodel access path. In general, the rule generator replaces a method that retrieves a collection element using a constant index with the constant index value in the metamodel access path.

The extent to which the generation of the mapping file can be automated depends on the transformation authoring framework and the representation of the input and output models that are being used. For example, if the models are represented using EMF, the mapping of metamodel elements to methods can be generated automatically, with no manual intervention by the user. However, in other standard or custom model-transformation frameworks, less automation may be possible, which would require the transformer author to provide the information manually. Similarly, in the RSA framework, the information about method to source/target mapping can be generated automatically. In other frameworks, such automation may not be possible.

3.4 Model Verification and Transform Comprehension

The transform rules inferred in Phase 1 of our approach can be used to support model verification and transform comprehension. Because the rules are stated in the metamodel vocabulary, they can be used in a straightforward manner to support these tasks.

Our approach distinguishes verification rules from querying rules. In a verification rule, the consequent is an exception that can be thrown if the antecedent is satisfied. In a querying rule, the consequent is an existential quantifier on an output metamodel access path; such a rule states that if the antecedent is satisfied, an element or property is created in the output model. The verification rules are used for checking whether a model is a valid input to a transformer, whereas querying rules are used for supporting general transform comprehension. We illustrate both of these use cases.

Model verification Given a set of verification rules Ψ for a transformer τ : M_I → M_O and an input model M_I, a verifier returns a subset of the rules in Ψ that are satisfied by M_I. If none of the rules are satisfied, M_I is a valid instance that the transformer can be executed on. However, if at least one of the verification rules is satisfied, M_I is not a valid input to the transformer; the transformer can fail with an exception when executed on M_I. To check whether a rule ψ ∈ Ψ is satisfied by M_I, the verifier finds the matching instances for the metamodel access path in the antecedent of ψ, and applies the condition stated in the antecedent rule predicate to the instances. If the condition is satisfied, M_I is an invalid input model to τ.

The verifier can be used in a batch mode in which it gives a list of matching rules and corresponding problematic input model elements. Alternatively, the verifier can be used in an interactive mode; in this mode, while the user is creating an input model, the verifier flags the problematic input model elements that could cause exceptions.

Transform comprehension A querying rule can be used to support user queries in a comprehension tool. For example, if an element that was expected in the output model is missing, the querying rules can be searched to find the ones that determine the creation of the missing element in the output model. These rules indicate the dependences of the missing output element to the input model elements. The user can then identify the cause by examining the input model elements to see whether they satisfy the rules, and correct the input model appropriately.

3.5 Discussion

In this section, we analytically evaluate our approach in terms of (1) the necessity and sufficiency of the constraints and transform rules, and (2) the extent of automation that is achievable for different steps of the approach. These aspects influence the usefulness and usability of the approach.

In general, static analysis cannot compute sufficient conditions for a statement to be reached, because of the presence of loops. Thus, a path constraint with respect to a path ρ = (s_1, ..., s_k) and a predicate γ is not a sufficient condition for γ to be true at s_k. However, model transformers are a very restricted class of Java programs. Transformers typically contain little or no business logic; much of the code is devoted to composing output model elements from input model elements, based on different conditions. Because of the limited ways in which transformers are coded, static analysis can attain far greater success in computing sufficient conditions for this class of programs than it might achieve for applications coded in less restricted ways. This obser-
viation is supported by our empirical results; almost all of the constraints that we examined manually for our subjects represented sufficient conditions. Another factor that affects the sufficiency of a constraint computed by our analysis is that the constraint is stated in terms of input variables only. Predicates on calls to external methods are removed from the constraints, as illustrated in Example 4; such predicates can affect whether the resulting constraint is sufficient for an exception to occur or an output element to be generated.

Although not sufficient, a path constraint is a necessary condition—if the constraint does not hold at $s_1$, $\gamma$ is definitely not true at $s_k$. An input constraint is a disjunction of path constraints. For an input constraint to be a necessary condition, the path constraint for each path must be computed. The presence of loops again makes this an unattainable goal.

Although a constraint may not be a sufficient condition, in our experience, most of the conditions on the input models that we examined manually in our empirical studies highlighted bugs that the user would want to fix. Thus, we think that the value of our approach need not be decided based on general arguments about sufficiency and necessity only, but, instead, be judged on the usefulness and relevance of the identified conditions. Even though a condition may not be a sufficient condition for a failure to occur, it does highlight problems in the input model that the user would want to address.

In terms of automation, some steps of the approach require user inputs—in the form of filters for the code analyzer and mapping information for the rule generator. However, these do not place an unreasonable enough burden on the transformer author to render the approach impractical. For analysis filters, our studies indicate that the use of only a few framework-specific filters can significantly reduce uninteresting constraints that provide no value. Moreover, the filters need to be specified only once, and, in our experience, can be identified with little effort. For rule generation, depending on the framework being used, some mapping information may need to be provided manually. But this information needs to be provided once and, in many cases, can be auto-generated. The verification and querying steps can be automated quite simply. Thus, overall, our approach achieves significant automation, and offers a solution that can help users in diagnosing efficiently the causes of failing or incomplete transformations.

4. Empirical Evaluation

To evaluate the feasibility and usefulness of our approach, we conducted two empirical studies. In the first study, we evaluated the accuracy of the analysis in terms of inference of useful constraints and transform rules. The second study was a user study, in which we investigated whether the use of transform rules can help users in diagnosing the causes of failing and incomplete transformations more efficiently.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Classes</th>
<th>Methods</th>
<th>Bytecode instructions</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subject-1</td>
<td>41</td>
<td>280</td>
<td>4904</td>
<td>285.5s</td>
</tr>
<tr>
<td>Subject-2</td>
<td>13</td>
<td>77</td>
<td>1212</td>
<td>15.7s</td>
</tr>
<tr>
<td>Subject-3</td>
<td>48</td>
<td>399</td>
<td>5340</td>
<td>280.5s</td>
</tr>
<tr>
<td>Subject-4</td>
<td>29</td>
<td>157</td>
<td>2449</td>
<td>33.08s</td>
</tr>
</tbody>
</table>

Table 2. Subjects used in the empirical evaluation.

4.1 Feasibility Study

In the first study, we evaluated the feasibility of our approach in terms of whether the approach can infer enough useful rules to support verification and comprehension tasks.

4.1.1 Experimental setup

We implemented the algorithm shown in Figure 9 using XYLEM. XYLEM uses the WALA analysis infrastructure\(^{10}\) to construct the call graph and the CFGs. XYLEM performs the analysis in two steps. In the first step, it performs points-to analysis, escape analysis, and control-dependence analysis. In the second step, it uses the results of the first step and computes exception constraints; we are currently implementing the computation of output constraints.

We used four experimental subjects; Table 2 lists these subjects along with information about the number of classes, methods, and bytecode instructions in each subject. These subjects are real model transformers that have been developed as part of ongoing research projects in IBM. All of the subjects was developed using RSA model-to-model transformation framework. Subject-1 and Subject-4 transformed a SOMA Service Model \(^{11}\) to a application specific E CORE model; these were intermediate models which were eventually transformed to different types of code artifacts. Subject-2 is the INFO TRANS transformer introduced in Section 2.1. Subject-3 transformed a SOMA Service Model to RSA Software Services Model \(^{12}\).

4.1.2 Goals and method

The goals of the study were to investigate (1) the number of constraints and rules identified by our approach, (2) the effectiveness of filters in removing uninteresting constraints, and (3) the extent to which duplicate rules are computed.

To compute the results, we ran XYLEM twice on each subject to compute exception constraints. After the first run, we asked the transformer authors to examine the computed constraints and identify filters that would remove invalid and bug constraints. As discussed in Section 3.2.2, invalid constraints cannot be true because of restrictions imposed by the transformer authoring framework, whereas bug constraints indicate potential transformer bugs, which, although interesting to the transformer author, are uninteresting from the transformer user’s perspective. We used the filters in the sec-

\(^{10}\)http://wala.sourceforge.net
\(^{12}\)http://www.ibm.com/developerworks/rational/library/05/510_gvc/
Table 3. Inferred exception constraints and verification rules.

<table>
<thead>
<tr>
<th>Exception Type</th>
<th>Subject</th>
<th>Number of Traversals</th>
<th>Initial Constraints</th>
<th>Final Constraints</th>
<th>Rules</th>
<th>Unique Rules</th>
<th>Rule Predicates</th>
<th>Access-path Length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Null-Pointer exception</td>
<td>Subject-1</td>
<td>1125</td>
<td>193</td>
<td>79</td>
<td>609</td>
<td>80</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Subject-2</td>
<td>321</td>
<td>19</td>
<td>19</td>
<td>1008</td>
<td>29</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Subject-3</td>
<td>1301</td>
<td>79</td>
<td>40</td>
<td>4576</td>
<td>174</td>
<td>1</td>
<td>2.6</td>
</tr>
<tr>
<td></td>
<td>Subject-4</td>
<td>688</td>
<td>49</td>
<td>21</td>
<td>34</td>
<td>23</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Class-cast exception</td>
<td>Subject-1</td>
<td>185</td>
<td>35</td>
<td>11</td>
<td>17</td>
<td>12</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Subject-2</td>
<td>30</td>
<td>3</td>
<td>2</td>
<td>36</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Subject-3</td>
<td>198</td>
<td>18</td>
<td>8</td>
<td>96</td>
<td>37</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Subject-4</td>
<td>71</td>
<td>23</td>
<td>6</td>
<td>12</td>
<td>9</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Array-index exception</td>
<td>Subject-1</td>
<td>12</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Subject-2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>36</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Subject-3</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Subject-4</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

ond run of XYLEM. We wrote a simple Java program to translate filtered exception constraints to verification rules and remove duplicate rules. For Subject-1 and Subject-2, the final rule set was examined by the transformer authors to determine the validity of the rules. All reported verification rules were found to be valid rules.

4.1.3 Results and analysis

Table 3 presents the results of the study. We show the data for the three types of exceptions separately, so that usefulness of each analysis is illustrated. Column 3 shows the number of traversals performed by XYLEM. The maximum number of traversals were performed for null-pointer exceptions (3435 traversals); the number of traversals for class-cast exceptions (484) and array-index exceptions (27) were much less. This is expected because dereference statements occur much more frequently in Java programs than typcast statements or collection-access statements of the form analyzed by XYLEM.

Columns 4 and 5 show, respectively, the number of initial constraints (computed in the first run) and the number of final constraints (computed in the second run in which filters were applied). The data illustrate that filters are effective in removing many uninteresting constraints. On average over all subjects and exceptions, the number of constraints was reduced by over 55%, from 429 initial constraints to 190 constraints, after filtering. The maximum reduction—over 60%—occurred for Subject-1. For Subject-3 and Subject-4, all constraints for array-index exceptions were filtered out.

Column 6 shows the number of verification rules that were translated from the final constraints. As mentioned in Section 3.3, the constraints are stated as a DNF formula over abstract predicates, whereas rules are stated as a conjunction of rule predicates. Thus, each disjunct, or path constraint, in an exception constraint gets translated as a verification rule. The data in column 6 indicate that the final constraints contained a large number of path constraints. For example, for Subject-3, 40 exception constraints for null-pointer exceptions resulted in 4576 rules—on average, 114 path constraints per exception constraint. There is a wide variation in the number of path constraints for the subjects: on average, Subject-3 had 97 path constraints, whereas Subject-4 had only 2 path constraints, per exception constraint; Subject-1 and Subject-2 had 7 and 47 path constraints, respectively, per exception constraint.

Column 7 illustrates that many duplicate rules occurred. For Subject-3, 4402 of the 4576 rules for null-pointer exceptions were duplicates; thus, after the removal of duplicate rules, only 174 rules remained. Over all subjects, the number of rules decreased from 6426 to 369 after the removal of duplicates—a reduction of over 99%.

Columns 8–10, show the minimum, maximum, and average number of rule predicates per transform rule. The data show that, typically, the transform rules are fairly simple in that the antecedent of the rules contains conjunctions of very few predicates. None of the rules, over all subjects, had more than six rule predicates in the antecedent.

Columns 11–12 show the minimum and maximum lengths of the metamodel access paths for the rules. An access-path length illustrates the chain of relations that occurs in a model, and, thus, is an indicator of the complexity of a metamodel. For our subjects, the maximum metamodel access-path length ranged from four to 10.

4.1.4 Discussion

Our study reveals several trends that illustrate the benefits of our approach. For our subjects, the approach inferred 369 useful rules, which is a significant number. The use of filters is essential because it can remove many uninteresting constraints; by doing so, it improves the effectiveness of model verification and transform comprehension. Moreover, many of the verification rules were duplicates; thus, removal of duplicate rules is an important step in our approach that is essential for improving its usability. The number of path constraints per exception constraint varied wide among our subjects—from 114 to two. The number of path constraints depends on the structure of the program and complexity of the input meta-model; therefore, the variation indicates that our subjects are structured quite differently, in terms of the
number of program paths and the input meta-model. The data also demonstrate the effectiveness of XYLEM in that it is able to analyze many paths.

4.2 User Study

Our second study was a user study, in which we tested the following hypothesis:

A user can perform the task of identifying and fixing bugs in an invalid input model more efficiently when guided by the transform rules than without the rules.

4.2.1 Experimental Setup

To select participants with different degrees of expertise, we identified the factors on which the expertise assessment could be based. Familiarity with MDD concepts is a key factor. We used INFOTRANS as the subject, which is created using the RSA transformation authoring framework. Thus, familiarity with the RSA capabilities for model creation, model browsing, and transformer authoring is another important factor. Finally, knowledge of code-navigation features provided by tools, such as Eclipse, is a factor that determines the efficiency with which a participant can navigate the transformer code to identify violated input model constraints. Based on these factors, we grouped the participants into three categories: expert (one participant, referred to as E1), typical (three participants, referred to as A1, A2, and A3), and novice (one participant, referred to as N1).

We created two debugging tasks: Task $T_1$, in which INFOTRANS fails with an exception, and Task $T_2$, in which INFOTRANS generates an incomplete output model. For each of the tasks, we created two sub-tasks, one in which the participants had to debug the problem without using the rules ($T_{wr}$), and another in which the participants had to debug a similar problem while guided by the transform rules ($T_r$). To enable a fair comparison of the effort required to complete the tasks, we ensured that each pair of subtasks $(T_{1,wr}, T_{1,r})$ and $(T_{2,wr}, T_{2,r})$ were of similar difficulty. We created four input models accordingly with errors, one each for $T_{1,wr}, T_{1,r}, T_{2,wr},$ and $T_{2,r}$.

For $(T_{1,wr})$ and $(T_{1,r})$, the participants were asked to fix input models; they were given access to the transformer code and were also allowed to use code debugging features. For the $(T_{1,wr})$ and $(T_{1,r})$, the participants were asked to fix the input models with the help of the rule-set only (with no access to the transformer code). Thus, we simulated the scenario in which transform users have to debug their models without needing to examine the transformer source code. The transform rules were created by running XYLEM on the INFOTRANS transformer; the computed rules were augmented with manually created querying rules (for output constraints).

We measured the time each user took to complete the task. During the study, the participants were allowed to ask questions about usage of the tools, but not on the input or output model instances.

4.2.2 Results and analysis

Table 4 lists the time taken by the participants to perform the tasks. As the table illustrates, all users—irrespective of their expertise levels—completed the tasks faster when they were guided by the rules than when they were not. For example, the expert participant took five minutes to complete the first task without rules and three minutes when using the rules. The participants with intermediate expertise took, on average, nine minutes to complete the first task without rules and only four minutes to complete it with rules. The novice participant took 16 and 14 minutes, respectively, to complete $T_{1,wr}$ and $T_{2,wr}$ and seven minutes each to complete $T_{1,r}$ and $T_{2,r}$. The maximum reduction occurred for user A1 for the second task. The user took 14 minutes to complete the tasks without rules and only two minutes to complete it with the rules. The minimum reduction occurred for user A3 for task $T_2$.

Figure 12 presents a different view of the data: it shows the percentage of time taken by each participant to complete the four tasks. As shown in the figure, the participants spent 62% to 78% of the total time in fixing models without the rules, whereas they spent significantly less time (22% to 38%) in fixing models using the rules.

In the feedback after the study, all participants mentioned that the transform rules were very useful in identifying and fixing the problems with the input models. They also felt that debugging transformers was different from debugging normal Java applications, as the inputs to transformers are typically more complex and have more elaborate syntax and semantics. Therefore, automated debugging support, that is customized for such applications can be useful; our ap-
approach provides such support. The participants unanimously wanted a visual representation of the rules for better usability. The novice participant suggested that the visual representation should return the matching input model elements for the rule, because of which transformation was failing. This would reduce the manual effort of performing the verification activity. One of the participants wanted a more interactive component that guides the user during model creation. Another user mentioned that a “self healing” or “recommendation” feature that suggested fixes for the invalid model elements would be very useful.

4.2.3 Discussion

Although our study is limited in nature, the results confirm our hypothesis that transform rules can enable a user to identify the cause of a failing transformation or an incomplete output model more efficiently. All the users found the rules useful, and each user performed the debugging tasks much faster when guided by the rules than when the rules were not used.

5. Related Work

MDD has been an active research area and there has been considerable work done in verification of model and testing of model transformations. In this section we present on how our approach differs from existing work.

Many researchers have worked on the verification of the transformation (e.g., (8; 14; 12)). They have tried to ensure the semantic equivalence between the input and output model of the transformation where the generated output model was code (8), or the transformation was of particular type, such as graph transformations (14), or the transformation was specified using some constraints (12). In our approach, we assume the transformation to be correct; we do not verify the transformation. The rules generated in our approach can be used by the user to understand the transformer, and reason out the cause for incomplete model generation and appropriately fix the issue in the input model.

Benoit Baudry et al. (2) have identified important issues that are required to be tackled to define sound and practical techniques for testing model transformation. Franck Fleurey et al. (5) present a general view of testing in different stages of MDD and a more detailed exploration of approaches to testing model transformation. They also highlight particular issues for the different testing tasks, including adequacy criteria, test oracles and automatic test data generation. Kuster et al. (11) address the problem of validation of transformers and propose techniques that can be used for generating test cases. We do not intend to test the transformers but assume the transformers to be correct and focus on developing approaches and techniques to enable users of transformers to understand the model transformation by treating the transformer as a black box.

Automated program-analysis techniques, such as program slicing (16) and symbolic execution (9), can be used to analyze source code to recover potential contracts, such as exception conditions (e.g., (4)). Automatically inferred contracts can assist service providers in creating semantic annotations and ensuring that the annotations are consistent (i.e., do not contradict the contracts present in code) and complete (i.e., do not miss contracts implicit in code). The Buse work is similar to ours. They locate exception throwing instructions and the symbolically track paths that lead to these exceptions. This symbolic execution generates predicates describing feasible paths yielding a boolean formula over program variables. This formula is further applied to generating human readable documentation.

Recently there has been a lot of work in the area of client checking wherein a client is supposed to follow the protocol provided by the implementer. Bierhoff and Aldrich (3) provide a solution for sound modular checking in the presence of aliasing that is based on typestate analysis. Two tools that need to be mentioned in the context of precondition generation are ESC/Java (6) and Boogie (1). ESC/Java implements essentially intraprocedural analysis and relies on user annotations to perform interprocedural analysis. Boogie does sound verification of Spec# specification but has a fairly restrictive ownership model.

6. Summary and Future Work

In this paper, we presented an approach for assisting users of model transformer in debugging their input models without examining the transformer code. The approach uses static code analysis to compute constraints on the input model under which a transformer could fail with an exception (exception constraints) or generate an incomplete output model (output constraints). The computed constraints are abstracted from code-level conditions to metamodel-level rules that can support model verification and transform comprehension. The verification rules can be used for checking whether a metamodel instance is a valid input to a transformer. The querying rules can help a user understand why an incomplete output model is generated.

Our empirical results indicate that the approach can be effective in computing a significant number of useful rules. We also conducted a user study to investigate how the inferred rules could enable transformer users to perform debugging and comprehension tasks more efficiently. All the participants in the study performed the debugging tasks faster with the rules than without them. These results suggest that our approach can be used to improve model-transformation tools by providing automated support for understanding transformations.

There are several interesting problems that future research could address.

Non-Java-based transformers In this paper, we focused on model-to-model transformers that are written in Java.
However, given that models are often represented using XML, XSLT is frequently used in practice for writing transformers. Future research could extend our approach to handle transformers implemented in XSLT (or, other transformer implementation technologies).

**Model-to-text transformations**  Model-to-model transformations are usually intermediate steps in MDD, with the end objective being to generate code (Java code, HTML pages, JavaScript code, etc.). Applying our approach to model-to-code—or, more generally, to model-to-text—transformations is an interesting direction for future research. Future work could identify the salient features in model-to-text transformations and explore how the static-analysis approach needs to be extended to generate useful rules.

**Interfaces for improving usability** In our current approach, the transform rules are presented as plain text to end users, who need to use simple or advanced search features to browse through the rules. The usability and comprehension of these rules can be significantly improved by developing intuitive graphical interfaces that are integrated with model-browsing capabilities. In [10] authors present an interesting extension to the usual debugging interfaces for Java programs. While debugging a program, a developer can select questions about program output from a set of “why did” and “why did not” queries that are derived using static and dynamic analyses. A similar kind of interface could be developed for understanding why elements are generated in output models.

**Improvements to code analysis** The static analysis performed by XYLEM could be improved to perform better analysis of array-index exceptions, collection classes, and include additional runtime exceptions. The analysis could also be improved to compute better constraints in the presence of calls to external methods. Currently, the conditions on return values from external method calls are inlined in the constraints, if the parameters of those calls have dependences on inputs. However, users might find such constraints difficult to understand especially if the transformer code is not available for inspection. Future improvements could also combine static analysis with dynamic information gathered from transformer executions to provide more accurate results to users. Another potential improvement is related to the computation of output constraints.

**References**


