Demystifying Model Transformations: An Approach Based on Automated Rule Inference

Mangala Gowri Nanda Senthil Mani Vibha Singhal Sinha Saurabh Sinha

IBM Research, India

{mgowri,sentmani,vibha.sinha,saurabhsinha}@in.ibm.com

Abstract

Model-driven development (MDD) is widely used to develop modern business applications. MDD involves creating models at different levels of abstractions. Starting with models of domain concepts, these abstractions are successively refined, using transforms, to design-level models and, eventually, code-level artifacts. Although many tools exist that support transform creation and verification, tools that help users in understanding and using transforms are rare. In this paper, we present an approach for assisting users in understanding model transformations and debugging their input models. We use automated program-analysis techniques to analyze the transform code and compute constraints under which a transformation may fail or be incomplete. These code-level constraints are mapped to the input model elements to generate model-level rules. The rules can be used to validate whether an input model violates transform constraints, and to support general user queries about a transformation. We have implemented the analysis in a tool called XYLEM. We present empirical results, which indicate that (1) our approach can be effective in inferring useful rules, and (2) the rules let users efficiently diagnose a failing transformation without examining the transform source code.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Debugging aids; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

General Terms Algorithms, Experimentation

Keywords Model-driven development, model-to-model transform, model validation, transformation comprehension, precondition analysis

OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA.

1. Introduction

Model-driven development (MDD) is a paradigm of software development that is based on the use of software modeling as the primary form of expression [10, 24]. It enables application design in terms of high-level "domain" concepts (from the problem space) instead of low-level "programming" concepts. A model is specified in a well-defined notation referred to as the metamodel. A model instance describes an actual system and conforms to the grammar of the metamodel. A transform[1] takes an instance of a model and converts it into another model (model to model transformation) or into code (model to code transformation). Typically, projects that follow an MDD methodology create a series of models at various levels of abstractions—that are successively refined—before actual code is created. As an example, a business analyst outlines the application work-flow as process models, which are captured in notations such as UML. The UML models are used to generate code skeletons to which developers add application logic. The conversion of a high-level model to a lower-level model or code can be done either manually or using automated transforms.

There are many factors that determine the effectiveness and efficiency of MDD. First, given a model transform, it must be verified that, for a valid input model, it generates the correct output model. To address this problem, existing research has developed many techniques for verifying and validating model transforms (*e.g.,* [1, 3, 9, 12, 16, 20]). Second, even if a transform is correct, the user of the transform might have to spend significant effort in creating a valid input model that does not violate transform assumptions. Unlike the first problem, the second problem has mostly been overlooked; this is the problem that we address in this paper.

1.1 Illustration of the Problem

Consider the ECORE[2] metamodel shown in Figure 1(a). The metamodel declares an element type `Attribute`, which has

[1] In this paper, we follow the terminology introduced by Baxter [2]. A *transform* is a function, or a program, that maps one model to another model (or text). A *transformation* is the application, or the execution, of a transform on a model instance.

[2] `http://www.eclipse.org/modeling/emf`

two features—name and type—neither of which is mandatory, as indicated by lowerBound = 0. Figure 1(b) presents an example model based on this metamodel. A Java transform is used to convert this model to a UML class; an excerpt of the code is shown in Figure 1(c). A user of this transform could potentially face the following problems while executing the transform with the input model illustrated in Figure 1(b).

EXAMPLE 1. *Transformation failure*. The transformation fails with a null-pointer exception at line 5, leaving the transform user with the task of going over the transform code to debug the failure. Going backwards from line 5, we see that type_src is assigned from attr.getType() at line 4 and attr is assigned from source at line 1. Therefore, by manual inspection of the code the user can infer that source.getType() should not be null. Further, based on knowledge of the framework used to create the transform, it is possible to infer that execute() is invoked only for model object instances of type Attribute. Therefore, the actual violated rule is that Attribute.getType() should not be null. The user fixes the problem by adding type = "" to the model, as shown in Figure 1(d). Moreover, to avoid this failure again with other input models, the user can add a rule to the metamodel to specify that type is a mandatory feature. For this example, the constraint is specified by adding lowerBound = 1 to the type feature in the metamodel, as shown in Figure 1(f). □

EXAMPLE 2. *Incomplete output model*. After the input model is fixed, the transformation runs to completion. However, in the generated output model, the UML property instance for the first Attribute instance does not have the type attribute. Once again, by manually inspecting the source code, the user determines that prop.setType() on line 7 generates the type attribute on the output Property instance—but only if the condition on line 5 evaluates true. Thus, the user infers the rule

$$\texttt{Attribute.getType().equals("String")} \wedge \texttt{UMLUtilities.findType(...)} \neq \texttt{null}$$

Applying this rule, the user modifies the Attribute instance to have type = "String" as shown in Figure 1(e). □

As illustrated in these examples, the user has to inspect manually the transform source code to identify the violated constraints that cause the transformation to fail or be incomplete. However, the source code of a transform is often not available for inspection by the transform user; or, if available, the users would prefer not to examine the code (which is usually written by someone else). Therefore, a debugging technique that supports users in understanding why a transformation failed or generated an incomplete output model, without requiring them to examine the transform source code, would be very useful.

Depending on the metamodel used to specify the input model, some of the simpler constraints could be specified

(a) Example metamodel definition

```
<eClassifiers xsi:type="ecore:EClass" name="Attribute">
  <eStructuralFeatures name="name" lowerBound="0" .../>
  <eStructuralFeatures name="type" lowerBound="0" ... />
</eClassifiers>
```

(b) A model defined using the metamodel

```
<Model>
  1: <Attribute name="id" />
  2: <Attribute name="name" type="String"/>
</Model>
```

(c) A Java transform for the metamodel

```
   public void execute( EObject source, EObject target ) {
1.   Attribute attr = (Attribute)source;
2.   Property prop = (Property)target;
3.   PrimitiveType ptype = null;
4.   String type_src = attr.getType();
5.   if (type_src.equals("String"))
6.     ptype = UMLUtilities.findType(...);
7.   if (ptype != null) prop.setType(ptype); }
```

(d) First correction to the model

```
<Model>
  1: <Attribute name="id" type="" />
  2: <Attribute name="name" type="String"/>
</Model>
```

(e) Second correction to the model

```
<Model>
  1: <Attribute name="id" type="String" />
  2: <Attribute name="name" type="String" />
</Model>
```

(f) Enhanced metamodel

```
<eClassifiers xsi:type="ecore:EClass" name="Attribute">
  <eStructuralFeatures name="name" lowerBound="0" .../>
  <eStructuralFeatures name="type" lowerBound="1" ... />
</eClassifiers>
```

Figure 1. An example metamodel, model, and Java transform.

in the metamodel definition by the transform author. More complex constraints could be documented in plain text or using a constraint language, such as OCL.[3] However, in either case, it is up to the transform author to maintain the constraints manually; a manual approach can cause the constraints to be incomplete, incorrect, and become outdated as the transform code evolves. Therefore, automated inference of transform constraints and mapping of constraints to rules is essential for developing a practical and effective debugging technique.

1.2 Overview of our Solution

In this paper, we present a new approach for assisting users in diagnosing the cause of a failed or an incomplete transformation. Overall, our goals are to support

- *Model validation*. We infer rules that state the conditions on the input model under which a transform fails; this addresses the problem illustrated in Example 1.
- *Transform comprehension*. We infer rules that state the conditions on input models under which a transform generates an incomplete output model; this addresses the problem illustrated in Example 2.

[3] http://www.omg.org/technology/documents/formal/ocl.htm

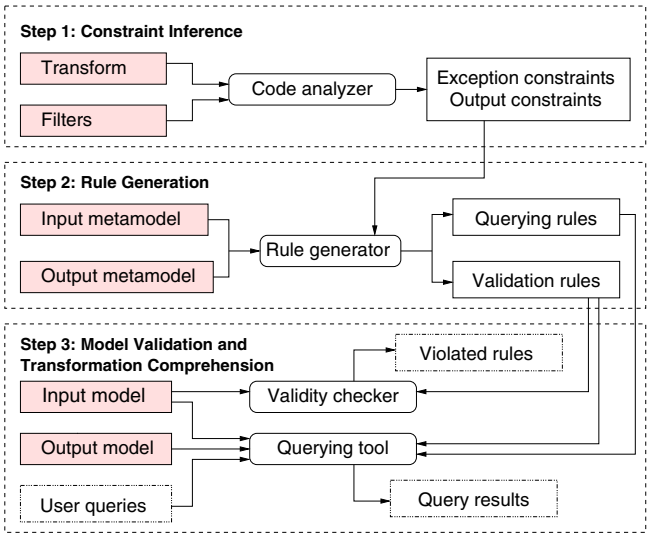

Figure 2. Overview of our solution.

Our approach consists of three steps; Figure 2 presents an overview of the steps.

In the *first step*, we analyze the transform code to extract automatically *exception constraints*, which can cause the transform to terminate with a runtime exception, and *output constraints*, which can cause the transform to generate an output model element. Because we are interested in only those constraints that can be mapped back into the input or output model, we first identify the variables in the transform code that map to the input and output model. For example in Figure 1(c) in the method:

```
public void execute(EObject source, EObject target)
```

`source` and `target` map to root elements in the input and output model, respectively. These entry points are manually identified and provided as inputs to our analysis.

At a potential exception-generating statement in the transform code, we identify the postconditions from which we derive the exception constraints. For example, at line 5 in Figure 1(c), we generate the postcondition $\langle$`type_src` $=$ `null`$\rangle$, which is the condition under which a null-pointer exception will occur. Starting from this postcondition, we compute weakest preconditions using a backward, interprocedural analysis. The preconditions generated are

$$\langle \texttt{source} \neq \texttt{null} \rangle \wedge \langle \texttt{source.getType()} = \texttt{null} \rangle$$

Finally, we check that the preconditions are "rooted" in the input model element (`source` in this example) before accepting them as valid exception constraints.

At each code point where the output model data structure is modified, we identify the postconditions from which we derive the output constraints. In Figure 1(c), for example, at line 7, we generate the postcondition $\langle$`ptype` $\neq$ `null`$\rangle$, which is the condition under which `prop.setType(ptype)` executes. Applying pointer and escape analysis [21], we further determine that `prop.setType(ptype)` "writes" to `prop.type`, which is an alias of `target.type`. Thus, a postcondition for an output constraint is the condition under which the transform code writes to an access path that is rooted in the `target`. The preconditions generated are

$$\langle \texttt{source.getType().equals("String")} \rangle \wedge \langle \texttt{UMLUtilities.findType(...)} \neq \texttt{null} \rangle$$

In the case of output constraints, we accept all preconditions that are rooted in input or output model elements (`source`, `target` respectively in this example). Preconditions containing other method calls with parameters that are in turn rooted in `source`, `target`, are also accepted.

All other constraints get filtered out. In practice, we also take as input a set of user-provided filters to remove uninteresting constraints. Constraints that do not fall into this pattern may be reported to the transform author as potential bugs in the transform code.

In the *second step*, we map the code-level constraints to metamodel-level rules. This is possible as all the constraints are rooted at the `source` or `target` elements, which are the elements that have been identified in step 1 as entry points into the input or output models. The mapping step basically raises the abstraction level of the code-level constraints so that they are stated in the language of the input metamodel and, therefore, are easier for the transform user to comprehend. A user-provided mapping file is used to translate the constraints to rules: exception constraints are mapped to *validation rules*, whereas output constraints are mapped to *comprehension rules*. Depending on the framework being used to write the transform, the generation of the mapping file can be automated to varying degrees.

In the *third step*, the metamodel-level rules may be used to construct a validity checker and a querying tool. The validity checker, given an input model for a transform, checks whether a model violates any of the validation rules. The querying tool can help the user understand the conditions on the input model elements under which an output model element is created; thus, the user can diagnose the cause of missing output-model elements.

Assumptions and Requirements Transforms can map a model to another model or to text (e.g., code). They can be implemented in an imperative manner or using declarative or rule-based languages. Our approach is applicable only to model-to-model transforms implemented in an imperative style. Our current implementation analyzes transforms written in the Java language. Our analysis works on the assumption that the entire input (and output) is captured in a single data structure that is mapped to the input (output) model. The input / output object may be passed in as a parameter or be constructed within the transform.

Our approach requires the identification of failure points in the code (for inferring exception constraints), and points at which output model elements are generated (for inferring output constraints). For Java, these program points can be identified automatically by identifying instructions that can

throw runtime exceptions, such as `NullPointerException`, or instructions that define the output object.

1.3 Contributions

The main benefit of our approach is that it provides automated support for diagnosing the cause of a failing or an incomplete transformation, without requiring an examination of the transform code.

Our analysis is similar to the computation of weakest preconditions (*e.g.,* [6, 8]). However, we apply the analysis to the domain of MDD, in which the inferred constraints are mapped to metamodel-level rules. Alternatively, such rules could be provided manually by the transform developer and used to annotate the model elements using a constraint-specification language. These rules could then be used to construct validity checkers. However, manual computation of rules may be time-consuming and error-prone [5].

Another important aspect of our work is the mapping of code-level constraints to metamodel-level rules, which has been recognized as an important feature affecting the usability of automatically inferred constraints [5]. Our tool partially automates these tasks, thereby reducing the burden on transform developers.

We implemented the solution for Java transforms and performed empirical evaluation using real applications. Our results indicate that, for the subjects considered, our approach can infer a significant number of useful rules. To validate this, we conducted a user study, in which we compared the efficiency of users in identifying and fixing problems with incorrect input models to a transform. 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.

The main contributions of the paper are

- The presentation of a static-analysis-based approach for inferring rules from model-to-model transforms and applying the rules to support model validation and transformation comprehension
- An implementation of the approach for transforms written in Java and models specified in EMF[2]
- Results of empirical studies, conducted using different types of models and transforms, that illustrate the benefits of the approach

The rest of the paper is organized as follows. In the next section, we introduce an example transform that we use to describe our solution in the remaining sections. In the subsequent three sections, we present the steps of our approach: constraint inference, rule generation, and validation/comprehension. Section 6 presents the empirical evaluation of our work. Section 7 discusses related work; finally, Section 8 summarizes the paper and lists directions for future research.

$\mathcal{E}$ = {DataModel, Artifact, ContextArtifact, Attribute, Annotation, ThisPackage}
$\mathcal{R}$ = {C: contains, I: inheritsFrom}
$\mathcal{P}$ = {name, type, value, multiplicity, isSimple}
δ_r : DataModel ↦ {((C,Artifact),artifacts,many), ((C,ContextArtifact),contextArtifacts,many),
Artifact ↦ {((C,Attribute),attributes,many)}
ContextArtifact ↦ {((C,Attribute),attributes,many)}
Attribute ↦ {((C,Annotation),annotations,many)}
Annotation ↦ ∅
ThisPackage ↦ {((I,Annotation),NA, NA)}
δ_p : DataModel ↦ {name}
Artifact ↦ {name}
ContextArtifact ↦ {name}
Attribute ↦ {name, type, isSimple, multiplicity}
Annotation ↦ {name, value}
ThisPackage ↦ {name, value}
e_r^t = DataModel

Figure 3. The input metamodel for INFOTRANS.

2. Definitions and Example

In this section, we present definitions and introduce an example model transform that we use to illustrate our approach.

2.1 Example

To illustrate the concepts described in this paper, we use an application called INFOTRANS[4] that takes as input a domain-specific information model, converts the model to a database schema, and creates a set of services that let users interact with the data. A domain subject-matter expert is expected to provide the input information model using a predefined metamodel. INFOTRANS converts the information model to a UML class model, using the model-to-model transformation framework provided by the Rational Software Architect (RSA).[5] Next, the class model is converted, using model-to-text transforms, to create a database-schema definition file and Java classes that implement the data services. For the purpose of this paper, we focus only on the model-to-model transformation part of INFOTRANS.

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.

DEFINITION 1. **(Metamodel)** *A* metamodel $\mathcal{M}$ *is a tuple* $(\mathcal{E}, \mathcal{R}, \mathcal{P}, \delta_r, \delta_p, e_r^t)$. $\mathcal{E}$ *a set of element types.* $\mathcal{R}$ *is a set of relation types.* $\mathcal{P}$ *is a set of properties.* $\delta_r : \mathcal{E} \rightarrow (\mathcal{R} \times \mathcal{E}, String, cardinality)$ *maps an element type to its related element types, where String is used as the representative name to declare the relationship; cardinality ('one', 'many', or 'NA') represents the allowed number of related element instances of that type.* $\delta_p : \mathcal{E} \rightarrow \mathscr{P}(\mathcal{P})$ *maps an element type to its associated properties.* $e_r^t \in \mathcal{E}$ *is the unique root element type.*

[4] INFOTRANS is a modified version of a real application that was developed in the context of a project at IBM Research.

[5] `http://www-01.ibm.com/software/awdtools/architect/swarchitect`

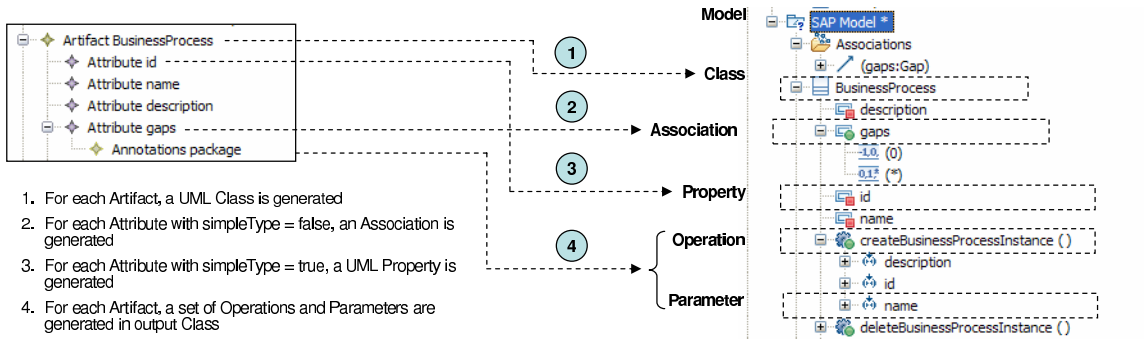

Figure 5. Mapping of some of the input model elements to output model elements (created by the transform for INFOTRANS).

```
<DataModel>
  <artifacts name="Gap">
    <attributes name="name" isSimple="true"
      type="String"/>
    <attributes name="resolution" isSimple="false"
      type="GapResolution">
      <ThisAnnotation name="current"/>
    </attributes>
  </artifacts>
  <contextArtifacts name="GapResolution">
    <attributes name="description" isSimple="true"
      type="String"/>
  </contextArtifacts>
</DataModel>
```

Figure 4. An input model (an instance of the metamodel shown in Figure 3) for INFOTRANS.

Examples of some commonly used schema languages for defining metamodels include ECORE,[2] XSD,[6] and MOF.[7] 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 3 shows the INFOTRANS input metamodel.[8] Artifact, Attribute etc. are examples of element types declared by the metamodel. The metamodel allows for two different types of relationships between object types; inheritance and containment. The function δ_r maps each element type to its relationship with other element types. The result of the mapping is a 3-tuple consisting of a pair of relation type and an element type, a name, and a cardinality. Artifact can contain multiple elements of instance Attribute and the relationship is identified with the name attributes. Except for the relationship between Annotation and ThisPackage, all other relationships are containment relations. Similarly, δ_p maps each element type to the properties associated with the element. For example, an Artifact can have a property called name.

DEFINITION 2. **(Metamodel access path)** *Let* $(\mathcal{E}, \mathcal{R}, \mathcal{P}, \delta_r, \delta_p, e_r^t)$ *be a metamodel. Let* $e\langle n\rangle$ *be the element with name* n*. A* metamodel access path π_{model} *is a sequence derived from the grammar*

$$
\begin{array}{lll}
\pi_{model} & ::= & e_r^t.\pi \\
\pi & ::= & n.\pi \mid \pi_{rec}.\pi \mid n \mid p \mid c \mid m(params) \\
\pi_{rec} & ::= & n.\pi_{rec} \mid (\pi_{rec})^+ \mid n \\
params & ::= & param, params \mid param \\
param & ::= & const \mid \pi_{model}
\end{array}
$$

where $e\langle n\rangle \in \mathcal{E}$; $p \in \mathcal{P}$; $n_i.n_j : e\langle n_j\rangle \in \delta_r(e\langle n_i\rangle)$; $n.p : p \in \delta_p(e\langle n\rangle)$; c *is an integer constant; and* m *is an external method that takes as input a list of parameters that may be constants or access paths.*

A metamodel access path is a sequence of containment relations that starts at the root element type and ends at

- An element name: DataModel.contextArtifacts.attributes
- A property type: DataModel.artifacts.attributes.isSimple
- An integer constant: DataModel.artifacts.0
- An external method: DataModel.contextArtifacts.name.equals("Current"), or UMLUtilities.findType(DataModel.contextArtifacts.name) where, equals is a standard external method from the Java API, UMLUtilities is a method from transform code that the user did not want to analyze.

Further, an access path can represent recursive model elements. Suppose, for example, in the metamodel of INFOTRANS, Attribute contained an Artifact called fact. Then, the metamodel could have a recursive access path that contains one or more occurrences of attributes.fact as follows: DataModel.artifacts.(attributes.fact)$^+$

Given a metamodel $\mathcal{M}$, a *model* can be constructed by creating instances of the element types, relations, and properties specified in $\mathcal{M}$. Figure 4 shows an input model, an instance of the metamodel (Figure 3), for INFOTRANS. The model has DataModel as the root element (e_r^t), which, in turn, contains instances of Artifact and ContextArtifact, identified by the relationship names artifacts and contextArtifacts, respectively. The Artifact instance with the property name whose value is Gap further contains an element of type Attribute, identified by the relationship name attributes. An Attribute instance has the following properties: name (with value resolution), type (with value GapResolution), and isSimple (with value false).

[6] http://www.w3.org/XML/Schema

[7] http://www.omg.org/technology/documents/formal/mof.htm

[8] This is a representation of the metamodel for the purposes of the paper only. The actual on-file representation varies, based on the framework used. For example, the ECORE syntax would look as shown in Figure 1(a).

```
    public void execute1( EObject source, EObject target ) {
 1.   Attribute attr = (Attribute)source;
 2.   Property prop = (Property)target;
 3.   PrimitiveType ptype = null;
 4.   org.eclipse.uml2.uml.Package umlprimitives =
 5.       UMLUtilities.loadPackage(URI.createURI(
 6.       Constants.UML_LIBRARY));
 7.   String type_src = attr.getType();
 8.   if (attr.isIsSimple()) {
 9.     if (type_src.equals("String"))
10.       ptype = UMLUtilities.findType(umlprimitives,"String");
     }
11.   if (ptype != null)
12.     prop.setType(ptype);
     }
    private void handleComplexType( Class cls, Iterator attrItr ) {
13.   Attribute attr = null;
14.   Property prop = null;
15.   Association assoc = null;
16.   while (attrItr.hasNext()) {
17.     attr = (Attribute) attrItr.next();
18.     if (!attr.isIsSimple()) {
19.       ThisPackage annotation =
20.           (ThisPackage)attr.getAnnotations().get(0);
21.       if (annotation != null) {
22.           prop = UMLUtilities.findProperty(cls, attr.getName())
23.   prop.setType(cls.getType());
    }
    public void execute2( EObject source, EObject target ) {
24.   DataModel sourceModel = (DataModel)source;
25.   Model targetModel = (Model)target;
26.   Iterator artifacts = sourceModel.getArtifacts();
27.   while (artifacts.hasNext()) {
28.     Artifact artifact = (Artifact)artifacts.next();
29.     Class cls = UMLUtilities.findClass(artifact.getName());
30.     handleComplexType(cls, artifact.getAttributes());
       ... } }
```

Examples of potential runtime exceptions:

1.	line 9	null-pointer exception	if `type_src` is null
2.	line 20	class-cast exception	if the first element in the list returned by `getAnnotations()` is not an instance of `ThisPackage`
3.	line 20	array-index exception	if `getAnnotations()` returns an empty list

Example of output statement:

4.	line 12	`prop.setType`	defines an output model element if `ptype` is not null, and the predicates in lines 8 and 9 evaluate true

Figure 6. Three methods from the Java transform code for INFOTRANS; three potential runtime exceptions that can occur, and the execution condition for an output statement.

2.3 Model-to-Model Transform

DEFINITION 3. **(Model-to-model transform)** *A* model-to-model transform $\tau : \mathcal{M}_I \rightarrow \mathcal{M}_O$ *is a program that given an input model* M_I *(an instance of metamodel* $\mathcal{M}_I$*) generates an output model* M_O *(an instance of metamodel* $\mathcal{M}_O$*).*

For the INFOTRANS application, the input ECORE model is converted to a UML class model using a transform. Figure 5 shows the input-to-output mappings for some of the model elements. For example, for each Artifact in the input model, the transform creates a Class, with a set of operations and parameters, in the output model.

Imperative transforms 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 transform is implemented in Java using the RSA transformation-authoring framework. Figure 6 shows the code fragments for three of the transform methods for INFOTRANS.

The bottom part of Figure 6 lists three potential exceptions that can occur during the execution of the transform. For example, 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. Our approach infers such conditions (validation rules) on input model elements under which the transform can fail with an exception. The rules can enable a user to diagnose efficiently the cause of the failure (*i.e.,* invalid input model elements), without having to examine the source code.

The figure also illustrates an example of conditions under which an output statement executes. Statement 12 defines an output model element. It is reached if, in the input model, an Attribute instance has property isSimple set to true and property type set to "String." Our approach infers such conditions (comprehension rules) under which an output model element is generated.

In the next three sections, we describe our solution in detail. In Section 3, we describe how we generate code constraints by analyzing the transform code. In Section 4, we describe how the code-level constraints are converted to metamodel-level rules. Finally, in Section 5, we describe the usage of inferred rules for input model validation and transformation comprehension.

3. Step 1: Constraint Inference

In Step 1 of our solution (Figure 2), we compute exception and output constraints from the transform code. In this section, we elaborate on the approach used to infer these constraints. The section is organized as follows. In Section 3.1, we formally define exception and output constraints. In Section 3.2, we present the algorithm for computing exception and output constraints. The analysis has been built on top of an existing tool XYLEM [22]. In Section 3.3, we explain XYLEM and the enhancements required to generate these constraints. In the following two sections, we explain some other intricacies of the analysis: how our path- sensitive algorithm overcomes the limitations of the pointer analysis (Section 3.4); and how we handle recursive access paths (Section 3.5). Finally, we discuss the soundness and completeness of our algorithm in Section 3.6.

3.1 Definitions

DEFINITION 4. **(Code access path)** *A* code access path π_{code} *is a sequence derived from the grammar*

$$
\begin{array}{ll}
\pi_{code} & ::= \pi | \pi_{code}.(\pi)^{+} \\
\pi & ::= v_{ref} | v_{ref}.\pi | m(params) | m(params).\pi \\
params & ::= param, params \mid param \\
param & ::= const \mid \pi_{code}
\end{array}
$$

$$
\begin{array}{rcll}
\gamma & ::= & \langle \pi_{code} \approx_{ref} \texttt{null} \rangle \mid & (1)\\
 & & \langle \pi_{code} \approx_{ref} \texttt{strConst} \rangle \mid & (2)\\
 & & \langle \pi_{code,1} \approx_{ref} \pi_{code,2} \rangle \mid & (3)\\
 & & \langle \pi_{code} \approx_{ref} \texttt{true} \rangle \mid & (4)\\
 & & \langle \pi_{code,1} \approx_{int} \pi_{code,2} \rangle \mid & (5)\\
 & & \langle \pi_{code} \approx_{int} \texttt{intConst} \rangle \mid & (6)\\
 & & \langle type(\pi_{code}) \approx_{type} \mathcal{T} \rangle \mid & *(7)\\
 & & \texttt{true} & (8)\\
 & & \neg\gamma & (9)\\
\approx_{ref} & ::= & = & \\
\approx_{type} & ::= & \in & \\
\approx_{int} & ::= & < \mid \leq \mid = &
\end{array}
$$

Figure 7. Predicates tracked by the analysis. $type(v)$ returns the type of a reference variable v. $\mathcal{T}$ is a set of types.

Typically, in Java, a code access path is defined as a variable or a variable followed by a sequence of field dereferences (`v.f1.f2...`). In our analysis, a field may be replaced by a method call—`v.f1.mx()`. This is similar to the access path defined by Buse [4]. However, unlike Buse's access paths that can have only simple methods with no parameters, in our analysis the method call could also have parameters, where the parameters are either constants or access paths—`v1.f1.mx().my("const", v2.mz())`. The method calls can be nested to any level. An example of a code access path with method calls is

```
source.getArtifacts().iterator().next().getName().equals(
source.getArtifacts().iterator().next().getType())
```

In our analysis, the code access path can also contain a set of repeated field dereferences or method calls in the presence of recursive model elements (explained in Section 3.5). An example of a code access path with repeated method calls is

```
source.(getArtifacts().iterator().next())+.getName()
```

A *predicate* γ, is a condition on one access path (unary predicate) or two (binary predicate) access paths. As shown in Figure 7, predicates 1 and 2 are unary reference and predicate 3 is a binary reference predicate; predicate 4 is a unary boolean predicate, and 5 and 6 are binary and unary integer predicates, respectively. Predicates 8 and 9 are standard nullary and negating predicates. Predicate 7 is a special predicate—on types of variables—that indicates the set of class types an access path may belong to. This is required to perform the analysis for class-cast exceptions (explained later in this section). A "unary" or a "binary" predicate may actually have multiple access paths if the predicate contains an access path with methods that have multiple non-constant parameters. An example predicate is

```
¬⟨source.getArtifacts().iterator().next().getName().equals(
source.getArtifacts().iterator().next().getType()) = true⟩
```

An *abstract state* Γ is a conjunction of predicates.

DEFINITION 5. **(Code constraint)** *Let $paths(s)$ be the set of paths from the entry statement s_e of a transform to statement s. Let γ be a predicate on a variable used at s. For a path, $\rho \in paths(s)$, let $\mathcal{C}(\rho, \gamma)$ be the state Γ_ρ at s_e such that if the predicates in Γ_ρ at s_e are true, then γ is true at s. Let $\mathcal{I} = \{i_1, \ldots, i_k\}, k \geq 1$, be the set of input variables to the program. A* code constraint *$\mathcal{C}_I(\gamma, s)$ is the disjunction $\bigvee_{\rho \in paths(s)} \Gamma_\rho[\mathcal{I}]$, where $\Gamma_\rho[\mathcal{I}]$ contains the predicates in Γ_ρ with respect to the variables in $\mathcal{I}$.*

A code constraint is a formula in the Disjunctive Normal Form (DNF), where each disjunct represents one program path from the entry of the program to the given statement s. Each program path is in turn represented as a conjunct of *predicates* that need to evaluate to true to be able to reach statement s. Consider the following code constraint:

$$
\begin{array}{c}
\langle \texttt{source.getName().equals(source.getType())} \rangle \wedge\\
\langle \texttt{source} \neq \texttt{null} \rangle \wedge\ \langle \texttt{source.getName()} \neq \texttt{null} \rangle \wedge\\
\langle \texttt{source.getProp()} = \texttt{null} \rangle\\
\vee\\
\neg\ \langle \texttt{source.getName().equals(source.getType())} \rangle \wedge\\
\langle \texttt{source} \neq \texttt{null} \rangle \wedge\ \ \langle \texttt{source.getName()} \neq \texttt{null} \rangle \wedge\\
\langle \texttt{source.getType()} = \texttt{null} \rangle
\end{array}
$$

Here, the first disjunct, represents the constraints obtained along one path, whereas the second disjunct (after the $\vee$), represents the constraints obtained along another path. In general, there may be an exponential number of paths. However, we compute only a fixed number of paths (based on a user-specified threshold) for each program point.

DEFINITION 6. **(Exception constraint)** *An* exception constraint *is a constraint $\mathcal{C}_{I(ex)}(\gamma, s)$, where γ represents the condition under which a runtime exception can occur at s in some execution of the program.*

An exception constraint can only contain code access paths rooted in the source model element (`source`). Moreover, for method calls in the access path, the parameters should be rooted at the source element, be a method call, or be a constant. An exception constraint whose access path does not satisfy these criteria is filtered away.

DEFINITION 7. **(Output constraint)** *Let v be an output variable of a transform τ. Let s_e be the exit statement of τ. Let $rdefs(s_e, v)$ be the reaching definitions of v.*[9] *An* output constraint *$\mathcal{C}_{out}(v)$ is the disjunction $\bigvee_{d \in rdefs(v)} = \mathcal{C}_I(\gamma_{cd}, s_{cd})$, where d is control dependent on (s_{cd}, L)*[10] *and γ_{cd} is the predicate asserting that the condition at s_{cd} evaluates to 'L', where L is either true or false.*

An output constraint can contain code access paths rooted in the source model element (`source`) and target model el-

[9] 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.

[10] 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.

```
algorithm TransformAnalysis
 input  τ          transform
 output C_I(ex)    exception constraints for τ
        C_I(out)   output constraints for τ
 global C_I        set of input constraints
begin
    // Identify exception constraints for null-pointer exceptions
 1. foreach statement s that dereferences v do
 2.     γ = ⟨v = null⟩; C_I = ∅
 3.     ComputeConstraints(s, {γ}); add C_I to C_I(ex)
    // Identify exception constraints for class-cast exceptions
 4. foreach typecast statement s : x = (T)y do
 5.     γ = ⟨type(x) ∋ (subtypes(T) ∪ {null})⟩; C_I = ∅
 6.     ComputeConstraints(s, {γ}); add C_I to C_I(ex)
    // Identify exception constraints for array-index exceptions
 7. foreach get statement s : c.get(intConst) do
 8.     γ = ⟨c ≠ null⟩; C_I = ∅
 9.     ComputeConstraints(s, {γ} ∪ {⟨c.size ≤ intConst⟩})
10.     if there is no statement on which s is directly/indirectly control
        dependent and that checks c.size then
11.         add C_I to C_I(ex)
    // Identify output constraints
12. foreach output variable v do
13.     foreach reaching definition d of v at the exit of τ do
14.         Let d be control dependent on (s_cd, L)
15.         Let γ be the predicate asserting that the condition at s_cd
                evaluates to L (true or false)
16.         ComputeConstraints(s_cd, {γ}); add C_I to C_I(out)
    // post-process
17.  remove non-input-variable predicates from C_I(ex) and C_I(out)
end
```

Figure 8. The analysis for computing exception and output constraints for a transform.

ement (`target`). Additionally, it can contain access paths rooted in library method calls (standard JDK or user-defined library classes). The method parameters could, in turn, be code access paths that satisfy the above criteria or be constants.

3.2 The Algorithm

Figure 8 presents the algorithm `TransformAnalysis` to compute input and output constraints for a given transform. The first step in the algorithm is the computation of appropriate postconditions.

Postconditions We check for three types of exceptions—Null Pointer Exceptions (NPE), Class Cast Exceptions (CCE) and Array Index Exceptions (AIE). These might generate exception constraints. However, the only exception constraints we are interested in are those that occur because of problems in the input elements. Additionally, we also check for conditions under which an output model element is written. These postconditions might generate an output constraint. The code variables that map to the input and output model elements are specified by the user as an input to the analysis. For ease of exposition, all through this paper we assume that the input model maps to a local parameter `source` and the output model to `target`.

- For null-pointer exceptions, at the dereference of a variable x, we define a postcondition $\langle$`x = null`$\rangle$ (lines 1–3)

```
procedure ComputeConstraints
 input  s          statement to start backward analysis from
        γ          stating predicate at s
 output C_I        constraints on input variables
 global CS         call stack of methods
        σ(s, Γ)    summary information at a call site s that maps an
                   incoming state Γ to a set of outgoing states
begin
 1. initialize state Γ to {γ}; initialize worklist with (s, Γ)
 2. while worklist ≠ ∅ do
 3.     remove (s, Γ) from worklist
 4.     foreach predecessor s_p of s do
 5.         if s_p is not the entry and not a call then
 6.             compute Γ′ for the transformation induced by s_p
 7.             if Γ′ is consistent then
 8.                 add (s_p, Γ′) to worklist if not visited
 9.         else if s_p is a call that invokes M then
10.             Γ_mx = map Γ to the exit of M
11.             Γ_me = σ(M, Γ_mx)
12.             if Γ_me = ∅ then // no summary exists
13.                 push M onto CS; analyze M with Γ_mx; pop CS
14.                 Γ_me = states at the entry of M
15.                 add Γ_me to σ(M, Γ_mx)
16.             Γ′ = map states in Γ_me to s_p
17.             add (s_p, Γ′) to worklist if not visited
18. if CS = ∅ then // method not being analyzed in a specific context
19.     if this is the entry method of τ then
20.         C_I = C_I ∨ Γ // add path constraint to DNF
21.     else foreach call site s_c that calls this method do
22.         Γ′ = map Γ to s_c; analyze caller starting at s_c with state Γ′
end
```

Figure 9. The new XYLEM analysis used to compute constraints on input variables under which the given predicate γ evaluates true at the given statement s.

- For class-cast exceptions, for a typecast statement `x = (T)y`, we define a postcondition $\neg\langle$`type(y)` $\in$ {`subtypes(T)` $\cup$ `null`}$\rangle$ which states that `y` is neither null nor of a type that can be cast to `T` (lines 4–6)
- For array-index exceptions, for a statement `c.get(const)`, we define a postcondition $\langle$`c` $\neq$ `null`$\rangle$ (lines 7–11)
- For output constraints, we first determine each statement s in the code where a field of `target` (or recursively, any field of a field of `target`) is written. If this write reaches the end of the transform (that is, the field is not overwritten), we find all conditionals that s is directly control dependent on. For each conditional C where s is control dependent on the true branch, we generate a postcondition C; similarly, for each conditional C where s is control dependent on the false branch, we generate a postcondition $\neg C$ (lines 12–17)

After computing the postconditions, we call procedure `ComputeConstraints`, which starts with the postcondition and executes a fix-point computation of the path-sensitive and context-sensitive analysis using XYLEM.

Fix-point computation and termination Given a predicate γ and an input statement s, `ComputeConstraints` computes the constraints on input variables under which γ evaluates to true at s. The algorithm for `ComputeConstraints` is presented in Figure 9.

	Statement	State transformation
(1)	`x = y`	$\Gamma' = \Gamma[x/y]$
(2)	`x = r.f`	$\Gamma' = \Gamma[x/r.f] \cup \{\langle r \neq \texttt{null}\rangle\}$
(3)	`if x op y`	$\Gamma' = \Gamma \cup \{\langle x\ op\ y\rangle\}$ (true branch)
		$\Gamma' = \Gamma \cup \{\langle \neg(x\ op\ y)\rangle\}$ (false branch)
(4)	`x = y op z`	$\Gamma' = \Gamma \setminus \Gamma[x]$
*(5)	`x = new T`	$\Gamma' = \Gamma \cup \{\langle x \neq \texttt{null}\rangle, \langle type(x) \in \{T\}\rangle\}$
*(6)	`x = (T)y`	$\Gamma' = \Gamma \cup \{\langle type(y) \in subtypes(T)\rangle\}$
*(7)	`x instanceof T`	$\Gamma' = \Gamma \cup \{\langle x \neq \texttt{null}\rangle,$ $\langle type(x) \in subtypes(T)\rangle\}$
*(8)	`x = r.m()` (ext)	$\Gamma' = \Gamma[x/r.m()]$
(9)	`x = r.m()` (app)	$\Gamma' = \sigma(r.m, \Gamma) \cup \{\langle r \neq \texttt{null}\rangle\}$

Figure 10. State transformations at some of the statements. Γ represents the state following a statement; Γ' represents the state preceding a statement.

`ComputeConstraints` essentially starts at the given statement s and works backward along the control flow graph (CFG) and applies a state transformation on each statement. The statement is followed by a consistency check to ensure that no conflicting predicates (such as, $\langle x = \texttt{null}\rangle$ and $\langle x \neq \texttt{null}\rangle$) have been generated. In case of conflict, that path is discarded.

`ComputeConstraints` abstracts away arithmetic expressions, which bounds the number of predicates that can be generated from arithmetic operations. The algorithm traverses a loop until the state no longer changes from one iteration to the next. Because, integer arithmetic over the loop induction variable is abstracted away, the analysis of a loop is bounded. The presence of recursive data structures can also cause an unbounded number of predicates to be generated. Our approach uses the standard method of k-limiting [17] to restrict the number of access paths that can be generated for recursive data structures. Section 3.5 illustrates in detail the processing of recursive model elements.

State transformation The analysis uses back substitutions to update state predicates. Figure 10 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 with y.

Since we are particularly interested in deriving access paths that are rooted in the `source` or `target` objects, the state transformations are geared towards generating extended access paths. Consider the state transformation at statement `x = r.f`. The updated state contains the predicates in the incoming state, with each occurrence of `x` in a predicate replaced with `r.f`, and predicate $\langle r \neq \texttt{null}\rangle$. Since we are substituting the left-hand-side of a computation by the right-hand-side, the generated predicate is precise for the given path except for recursive paths. This works correctly as the analysis works on an SSA language representation where each *use* of a variable has exactly one definition.

In Figure 10 transformation 8 is related to the code access-path representation that can contain nested method calls. 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.

Interprocedural path exploration To perform efficient interprocedural analysis, the algorithm computes method summaries. The summary σ for method M maps a state Γ at the exit of M to a set of states $\Gamma'_1, \ldots, \Gamma'_n (n \geq 1)$ at the entry of M, where each Γ'_i represents the transformation of Γ along a path in M.

When the analysis (Figure 9), reaches a call site to M, it maps Γ to the exit of M (state Γ_{mx}) and reuses a summary if it exists (line 16). If not, the algorithm descends into the called methods to analyze them (line 13). It uses a call stack to ensure a context-sensitive processing of called methods.[11] After analyzing the called method, the algorithm saves the summary information for reuse in subsequent traversals (lines 14–15).

On reaching the entry of the method that is not being analyzed in a specific context (line 18), the algorithm ascends to all call sites that call the method (lines 21–22). If the entry of the transform is reached, the algorithm adds the state predicates as a disjunct to the input constraints (lines 19–20).

Example 4 gives an example of how to generate access paths across method boundaries.

Null-pointer exceptions To compute constraints for potential null-pointer exceptions, `TransformAnalysis` processes each statement in the transform 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 γ to $\langle x = null\rangle$; then, it calls procedure `ComputeConstraints`, which computes the conditions on input variables under which γ evaluates true at s.

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

State transformations 5–7 shown in Figure 10 are relevant for the analysis of class-cast exceptions. For example, transformation 7 (that occurs at a statement `x instanceof T`) adds two predicates to the incoming state Γ: the first predicate asserts that `x` cannot be null because, in the Java semantics, a `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 $\langle \texttt{x} \neq \texttt{null}\rangle$ to the state because, in Java, a `null` can be cast to any type.

[11] 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.

	Exception Constraint	Validation Rule
1.	⟨source.isIsSimple() = true⟩ ∧ ⟨source.getType() = null⟩	(DataModel.artifacts.attributes.isSimple = true) ∧ (DataModel.artifacts.attributes.type = null) ⟹ NPE (DataModel.contextArtifacts.attributes.isSimple = true) ∧ (DataModel.contextArtifacts.attributes.type = null) ⟹ NPE
2.	⟨source.getArtifacts()*.getAttributes()*.isIsSimple() ≠ true⟩ ∧ ⟨*type*(source.getArtifacts()*.getAttributes()*.getAnnotations().get(0)⟩ ∋ {ThisPackage})	(DataModel.artifacts.attributes.isSimple ≠ true) ∧ (*type*(Datamodel.artifacts.attributes.annotations.0) ∋ {ThisPackage}) ⟹ CCE
3.	⟨source.getArtifacts()*.getAttributes()*.isIsSimple() ≠ true⟩ ∧ ⟨source.getArtifacts()*.getAttributes()*.getAnnotations().size ≤ 0⟩	(DataModel.artifacts.attributes.isSimple ≠ true) ∧ (Datamodel.artifacts.attributes.annotations.size ≤ 0) ⟹ AIE
	Output Constraint	**Comprehension Rule**
4.	⟨source.isIsSimple() = true⟩ ∧ ⟨source.getType().equals("String") = true⟩ ∧ ⟨UMLUtilities.findType(...) ≠ null⟩	(DataModel.artifacts.attributes.isSimple = true) ∧ (DataModel.artifacts.attributes.type.equals("String") ⟹ ∃ Model.classes.properties.type (DataModel.contextArtifacts.attributes.isSimple = true) ∧ (DataModel.contextArtifacts.attributes.type.equals("String") ⟹ ∃ Model.classes.properties.type

Table 1. The exception and output constraints and the corresponding validation and comprehension rules inferred for the three exceptions and output statement in the INFOTRANS code fragment (Figure 6). The first constraint causes two transform rules to be generated, whereas the other constraints result in one rule each. The "*" in the exception constraints represents ".next()".

Given two type predicates on a variable v, they are resolved as

$$\langle type(s) \in \mathcal{T}_1 \rangle \wedge \langle type(s) \in \mathcal{T}_2 \rangle = \begin{array}{ll} \text{if } \mathcal{T}_1 \cap \mathcal{T}_2 = \emptyset, & \text{if } (\mathcal{T}_1 \cap \mathcal{T}_2) \neq \emptyset \\ \text{conflict,} & \text{otherwise} \end{array}$$

EXAMPLE 3. Consider the program fragment

```
public class A {}
public class A1 extends A {}
public class A2 extends A {}
public class A21 extends A2 {}
public class A22 extends A2 {}
[1] A a = new A1();
[2] if ( a instanceof A2 ) {
[3]     A21 a21 = (A21)a;
```

To determine whether a class-cast exception can occur at line 3, the procedure ComputeConstraints is invoked with predicates $\langle a \neq \texttt{null} \rangle$ and $\neg\langle type(a) \in \{\texttt{A21}\} \rangle$. Statement 2 generates the predicates $\langle a \neq \texttt{null} \rangle$ and $\langle type(a) \in \{\texttt{A2,A21,A22}\} \rangle$. The resolved set of predicates now consists of $\gamma_1 = \langle a \neq \texttt{null} \rangle$ and $\gamma_2 = \langle type(a) \in \{\texttt{A2,A22}\} \rangle$. Then, statement 1 generates predicate $\langle type(a) \in \{\texttt{A1}\} \rangle$, which is inconsistent with γ_2. Therefore, the path (1, 2, 3) is infeasible and, consequently, no class-cast exception can occur at line 3. □

Array-index exceptions We perform a limited analysis of statements, such as statement 20 in the INFOTRANS code (Figure 6), 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. We compute the conditions on input variables under which (1) s is reached with ⟨c≠null⟩, and (2) and there is no condition that checks the size of c on which s is directly or transitively control dependent.

EXAMPLE 4. For the call to get() at statement 20 in Figure 6, the analysis calls ComputeConstraints with the initial state containing $\gamma_1 = \langle \texttt{attr.getAnnotations()} \neq \texttt{null} \rangle \wedge \langle \texttt{attr} \neq \texttt{null} \rangle$. At statement 18, it picks up predicate $\gamma_2 = \langle \texttt{attr.isIsSimple()} \neq \texttt{true} \rangle$. Statement 17 updates both the predicates by replacing attr with attrItr.next(). Next, at statement 16, the analysis adds the predicate $\gamma_3 = \langle \texttt{attrItr.hasNext()} = \texttt{true} \rangle$. At line 13, $\langle \texttt{attr} = \texttt{null} \rangle$ is not added as it conflicts with $\langle \texttt{attr} \neq \texttt{null} \rangle$.

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 predicates by replacing attrItr with artifact.getAttributes(). Thus, at this point γ_1 is

$$\langle \texttt{artifact.getAttributes().next().getAnnotations()} \neq \texttt{null} \rangle \wedge \langle \texttt{artifact.getAttributes()} \neq \texttt{null} \rangle$$

Continuing in the same manner through statements 28, 27, 26, and 24, the analysis computes γ_1 at entry as

$$\langle \texttt{source.getArtifacts().next().getAttributes().next().getAnnotations()} \neq \texttt{null} \rangle \wedge \ldots$$

Predicate γ_2 and γ_3 are updated similarly.

Next the algorithm checks if statement 20 is control dependent on any statement $v.size$ ***op*** intConst where v is an alias of getAnnotations() and ***op*** is one of $\{<, \leq, =\}$. The algorithm 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 on *unchecked access to an array location*. To aid in rule generation, we now convert γ_1 into the more meaningful predicate

$$\langle \texttt{source.getArtifacts().next().getAttributes().next().getAnnotations().size()} \leq 0 \rangle$$

We filter out predicates like ⟨source.getArtifacts() ≠ null⟩ that are implicit. □

Column 2 of Table 1 shows the constraints inferred for the three INFOTRANS exceptions (Figure 6). For brevity, we

have replaced occurrences of "`.next()`" in the constraints with "`*`" in the table.

Output constraints Intuitively, an output constraint captures the conditions on input variables under which an output element is generated. However, once the postconditions have been identified, the rest of the computation is similar to that of exception constraints.

EXAMPLE 5. Consider statement 12 in the INFOTRANS code fragment (Figure 6), 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 $\langle$`ptype` $\neq$ `null`$\rangle$. At statement 10, `ptype` is replaced with the external method call to `UMLUtilities.findType(...)`. Next, the analysis picks up predicate $\langle$`type_src.equals("String") = true`$\rangle$ at statement 9 and $\langle$`attr.isIsSimple() = true`$\rangle$ at statement 8. After processing the assignments at statements 7 and 1, the analysis identifies the following conditions at the entry of the method

$\langle$`source.isIsSimple() = true`$\rangle \wedge$
$\langle$`source.getType().equals("String") = true`$\rangle \wedge$
$\langle$`UMLUtilities.findType(...)` $\neq$ `null`$\rangle$ □

Constraint Filtering Our approach uses filters to improve the accuracy of the computed constraints. The filters consist of "invalid constraints" and "bug constraints." The first category filters predicates that cannot be true because of constraints imposed by the transformation-authoring framework. 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 `list.f` of such a list, predicate $\langle$`list` $\neq$ *null*$\rangle$ need not be generated. Our studies (Section 6.1) indicate that the use of only a few framework-specific filters can improve the results of the analysis significantly by removing many false predicates. Moreover, the filters need to be specified only once, and, in our experience, can be identified with little effort.

The second category filters out constraints that indicate potential bugs in the transform code. Such constraints, although relevant for the transform author, are uninteresting from the transform user's perspective. In fact, the transform author would either fix the potential bugs or filter out the constraints before computing the transform rules.

EXAMPLE 6. Consider the following example

```
[1] x = null;
[2] if ( getTransformType().equals("f1") )
[3]     x = new T("1");
[4] else if ( getTransformType().equals("f2") )
[5]     x = new T("2");
[6] x.foo()
```

The dereference of `x` at line 6 is a potential null-pointer exception. However, the transform author may know that `getTransformType()` always returns `"f1"` or `"f2"`. In this case, we need to add a filter so as not to generate an unnecessary rule. If `getTransformType()` may return other strings, it is the author's responsibility to fix the bug. □

3.3 Implementation over XYLEM

To compute the constraints, we leverage the null-dereference analysis implemented in the XYLEM tool [22]. The goal of the XYLEM analysis is to identify a program path along which a dereferenced variable can be null. Starting at a statement s_r that dereferences variable v, XYLEM performs a backward, path-sensitive and context-sensitive analysis to identify such a path.

While the basic infrastructure of backward, path-sensitive and context-sensitive analysis remains the same, the driver of the analysis is completely new. The current analysis drives the paths through library methods and tries to reach the top of the call graph. The older analysis used heuristics to handle library methods and aimed to stop at any appropriate method boundary.

The statement transformations have also been modified to support generation of access paths. So rather than use symbolic heap locations, we build the entire access path by concatenating individual access paths. Since this may generate an exponential number of paths, we limit the number of paths we explore using a user-defined threshold.

In addition, we made several extensions and enhancements to the original analysis: (1) the analysis computes a form of access paths that can contain nested method calls and parameters to those method calls; (2) in addition to identifying null-pointer exceptions, the extended analysis identifies potential class-cast exceptions 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.

3.4 Pointer Analysis and Aliasing

The accuracy of pointer analysis affects the number of computed constraints: a less accurate analysis would cause more spurious constraints to be computed, which could affect the usefulness of the approach. Our implementation uses a flow and context-sensitive pointer analysis [21]. The lack of path-sensitivity in the pointer analysis is made up for by the path-sensitive XYLEM propagation. To illustrate the effects of pointer analysis, consider program (a) below:

```
[1] if ( src.type.val )          [1] !<src.type.val>
[2]     x = src.attr.prop1;      [2]
[3] else                         [3]
[4]     x = src.attr.prop2;      [4] <src.attr.prop2.g1 =
                                      null>
[5] if ( src.type.val )          [5] !<src.type.val>
[6]     y = x.g2;                [6]
[7] else                         [7]
[8]     y = x.g1;                [8] <x.g1 = null>
[9] y.foo();                     [9] <y = null>
          (a)                              (b)
```

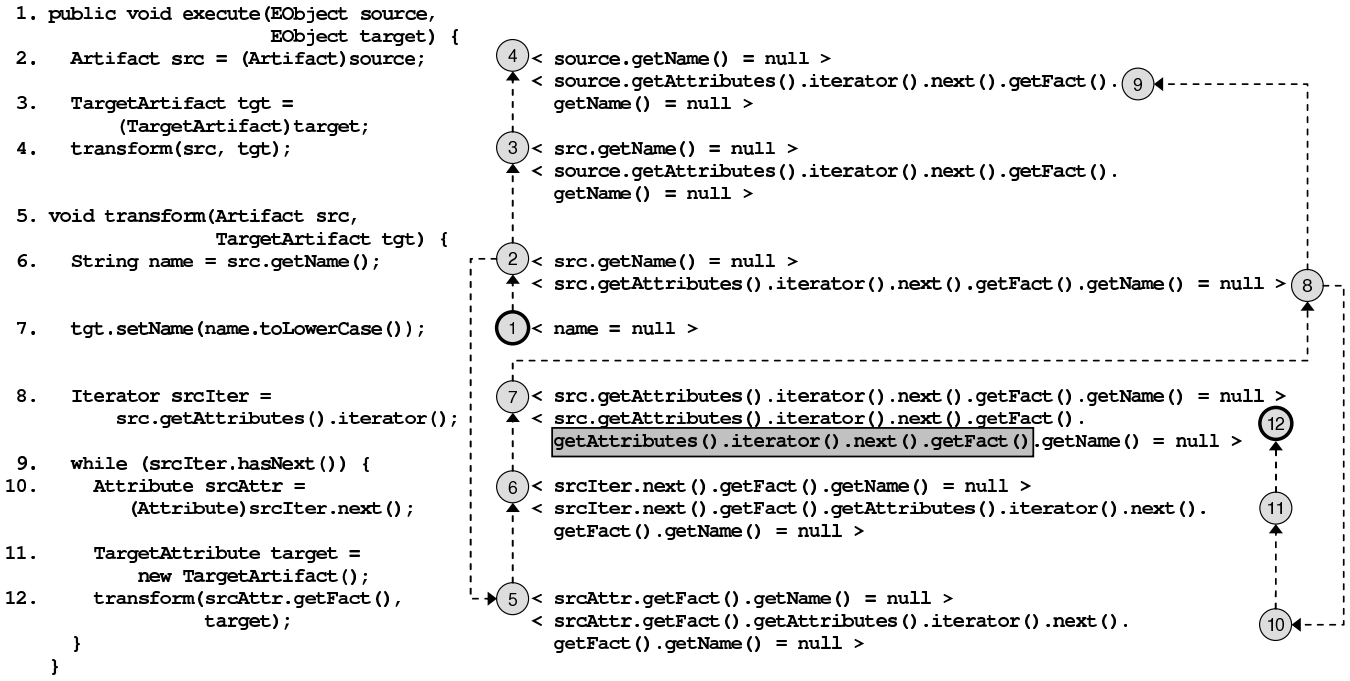

Figure 11. Example to illustrate the processing of recursive model elements by our approach. The figure shows the order in which the predicates are generated, starting at statement 7.

A path-sensitive pointer analysis would identify that at line 6, `x` can point to `src.attr.prop1` only, whereas at line 8, `x` can point to `src.attr.prop2` only. Using such precise points-to information, our analysis would compute two constraints for the dereference of `y` at line 9

$$(\langle \texttt{src.type.val} = \texttt{true} \rangle \wedge \langle \texttt{src.attr.prop1.g2} = \texttt{null} \rangle) \vee$$
$$(\langle \texttt{src.type.val} = \texttt{false} \rangle \wedge \langle \texttt{src.attr.prop2.g1} = \texttt{null} \rangle)$$

However a path-insensitive pointer analysis does not take into account branch correlation and assumes that at lines 6 and 8, `x` may point to either `src.type.prop1` or `src.type.prop2`. This less accurate points-to information combined with the older XYLEM analysis would cause our analysis to compute two additional, spurious constraints

$$(\langle \texttt{src.type.val} = \texttt{true} \rangle \wedge \langle \texttt{src.attr.prop2.g2} = \texttt{null} \rangle) \vee$$
$$(\langle \texttt{src.type.val} = \texttt{false} \rangle \wedge \langle \texttt{src.attr.prop1.g1} = \texttt{null} \rangle)$$

There are two factors that help make our analysis precise—(1) we generates constraints on extended access paths rather than on precomputed points-to information, and (2) the predicate propagation is path sensitive. By the first criterion, we build access paths by replacing the left-hand-side of a computation by the right-hand-side—which makes it precise for the given path; and by the second criterion, we ensure that only valid paths are traversed. Thus, in the example above, the analysis ensures that if the predicate $\langle$`src.type.val = true`$\rangle$, then the path traversed is through the lines $1, 2, 5, 6, 9$ and similarly for $\langle$`src.type.val = false`$\rangle$, then the path traversed is through the lines $1, 4, 5, 8, 9$ and thus computes only the correct constraints. Part (b) of the figure above shows the predicates generated along the path through the lines $9, 8, 5, 4, 1$.

3.5 Recursive Model Elements

To illustrate the processing of recursive model elements, consider the transform code shown in Figure 11. The input model of the transform has recursive elements, as shown by the following containment relations

Model	$\mapsto$ {(C,Artifact,rootArtifact,one)}
Artifact	$\mapsto$ {(C,Attribute,attributes,many)}
Attribute	$\mapsto$ {(C,Artifact,fact,one)}

The program contains a recursive method `transform()` (line 5), which is called at line 4 from the entry method `execute()` and recursively at line 12. The right side of the figure shows the state predicates that are propagated by the analysis. The numbers next to the predicates indicate the order in which the predicates are generated.

Suppose that the dereference of `name` at line 7 could cause a null-pointer exception. The analysis starts at line 7 with predicate $\langle$`name = null`$\rangle$. Using the standard back substitution, the predicate gets transformed to $\langle$`src.getName() = null`$\rangle$ at line 6. At the entry of `transform()`, the predicate is propagated, after mapping to actual parameters, to call site 4, and finally to the entry of `execute()` (as predicate 4).

At call site 12, the mapped predicate becomes $\langle$`srcAttr.getFact().getName() = null`$\rangle$ (predicate 5). Statements 10 and 8 transform the predicate to $\langle$`src.getAttributes().iterator().next().getFact().getName() = null`$\rangle$ (pred-

icate 7), which reaches the entry of `transform()`. This is the end of iteration 1 of the algorithm.

In the next iteration, predicate 8 is propagated to call sites 4 and 12. At call site 12, the mapped predicate 10 is propagated to line 10, where predicate 10 is transformed to ⟨`srcIter.next().getFact().getAttributes().iterator().next().getFact().getName() = null`⟩ (predicate 11). Finally, at line 8, the analysis computes predicate 12, whose access path has the repeating sub-sequence `getAttributes().iterator().next().getFact()`, illustrated in the figure by the shaded portion of predicate 12.[12] We mark off the recursive parts of the access path and propagate only the folded structure. Thus, the recursive sub-sequence is folded to generate predicate ⟨`src.getAttributes().iterator().next().getFact().getName() = null`⟩, which is the same as predicate 7 that was computed at line 8 in the previous iteration. Thus, the analysis finds no new predicates and hence terminates.

3.6 Soundness and Completeness

We now evaluate the soundness and completeness of our analysis for computing code constraints. Recall from Definition 5 that a code constraint $\mathcal{C}_I(\gamma, s)$ is a disjunction of constraints along the paths to statement s.

An *incomplete analysis* could either fail to compute $\mathcal{C}_I(\gamma, s)$, or fail to compute a disjunct (*i.e.,* a path constraint) for $\mathcal{C}_I(\gamma, s)$, for statement s. The first type of incompleteness can occur because our current implementation analyzes only three types of runtime exceptions; a transform could fail because of a runtime exception type, such as `ArrayStoreException`, not currently handled by the analysis. The analysis handles only limited forms of array-index exceptions. Moreover, a transform could fail because of exceptions thrown by calls to external (*e.g.,* JDK API) methods; our implementation does not compute constraints for such exceptions. The second type of incompleteness can occur because the analysis along a path can abort. Our algorithm uses three parameters to bound the analysis: the time required to analyze a path, the state size, and the number of paths through a method [22]. If the upper bounds for these parameters are reached, the algorithm can miss computing some path constraints. However, in our empirical studies (Section 6.1), this did not occur for any of the subjects.

An *unsound* analysis could compute spurious input constraints for a statement (*i.e.,* constraints that cannot be satisfied in any execution). The sources of unsoundness include limitations of static analysis in processing loops and arithmetic expressions; and, as illustrated earlier, imprecision in points-to analysis.

Related to the discussion of soundness and completeness is whether a path constraint in $\mathcal{C}_I(\gamma, s)$ represents a necessary and/or sufficient condition (or neither). The factors that introduce unsoundness can also cause a path constraint to not be a sufficient condition. However, a path constraint is a necessary condition: if an input object does not satisfy a path constraint to a statement s, the relevant behavior (failure or output generation) cannot occur at s along the path.

[12] To identify a recursive sub-sequence in an access path, we ignore the actual parameters of any method calls that appear in the path; we use only the signatures of these methods.

[RULEPREDICATE]

$$
\begin{array}{lll}
\psi & ::= & [\pi_{model} \approx_{ref} \texttt{null}] \mid \\
 & & [\pi_{model} \approx_{ref} \texttt{strConst}] \mid \\
 & & [\pi_{model} \approx_{ref} \texttt{true}] \\
 & & [\pi_{model}.size \approx_{int} \texttt{intConst}] \mid \\
 & & [\pi_{model} \approx_{type} \mathcal{T}] \mid \qquad \mathcal{T} \subseteq \mathcal{E} \\
\approx_{ref} & ::= & = \mid \neq \\
\approx_{int} & ::= & < \mid \leq \mid = \mid \neq \mid > \mid \geq \\
\approx_{type} & ::= & \in \mid \ni
\end{array}
$$

[TRANSFORMRULE]

$$
\begin{array}{lll}
\Psi & ::= & \psi_1 \wedge \ldots \wedge \psi_k \implies result \qquad k \geq 1 \\
result & ::= & excp \mid \exists(\pi_{model}) \\
excp & ::= & \texttt{NPE} \mid \texttt{CCE} \mid \texttt{AIE}
\end{array}
$$

Figure 12. Rule predicates defined with respect to metamodel access paths (top). Transform rules defined with respect to a pair of input and output metamodels (bottom).

4. Step 2: Rule Generation

Step 2 of our approach (Figure 2) converts code-level constraints to model-level rules.

DEFINITION 8. **(Rule predicate)** *A* rule predicate ψ*, defined with respect to an access path in a metamodel* $\mathcal{M}$*, is a predicate of the form shown in Figure 12.*

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

DEFINITION 9. **(Transform rule)** *A* transform rule Ψ *defined with respect to a transform* $\tau : \mathcal{M}_I \rightarrow \mathcal{M}_O$ *is a rule, of the form shown in Figure 12. 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 INFOTRANS. 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 transform. Exception constraints are mapped to validation rules, whereas output constraints are mapped to comprehension rules.

Note that whereas a code constraint is a DNF formula (a disjunction of conjunctions) over predicates, a transform rule is a conjunction of rule predicates. The rule generator trans-

lates each conjunct (or, a path constraint) in a code-level 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 transform. 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 the same consequent (NPE); such duplicate rules are removed during rule generation.

The constraint-to-rule translation requires converting a code predicate γ to a rule predicate ψ, which, in turn, essentially involves converting a code access path π_{code} to a metamodel access path π_{model}. Thus, the core of the rule-generator component is the access-path translation step. Recall from Definitions 4 and 2 that π_{code} is a sequence of dereferences of variables or method return values, whereas π_{model} is a sequence of containment relations that is composed of metamodel element names and method names and possibly ending with a property. For each reference variable v or method $m()$ in π_{code}, the rule generator has to identify the metamodel element 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 13 shows a partial XML representation of the mapping information for INFOTRANS.[13] A mapping file, in general, links metamodel element names and properties to entities in the transform inputs. For INFOTRANS, this requires linking the input ECORE metamodel element names and properties to Java methods and fields.

In the representation shown in Figure 13, each metaModelElement entry maps a metamodel element name or property type to Java method names. (Although not illustrated in this example, the mapping file represents methods by their signatures, which can accommodate overloaded and overridden methods.) 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 transform—a method that is invoked to perform a transformation. (In the RSA transformation-authoring framework, a transform can have multiple main methods; this may not be true for other frameworks.) For example, method `execute1()` takes as

[13] Figure 13 presents only the information that is required for mapping the exception and output constraints shown in Table 1 to transform rules.

```
<mapping>
   <metaModelElement name="isSimple">
      <method name="isIsSimple"/>
   </meta-model-element>
   <metaModelElement name="type">
      <method name="getType"/>
      <method name="setType"/>
   </meta-model-element>
   <metaModelElement name="artifacts">
      <method name="getArtiacts"/>
      <method name="setArtifacts"/>
   </meta-model-element>
   <metaModelElement name="attributes">
      <method name="getAttributes"/>
      <method name="setAttributes"/>
   </meta-model-element>
   <metaModelElement name="annotations">
      <method name="getAnnotations"/>
      <method name="setAnnotations"/>
   </meta-model-element>

   <methodName name="execute1()"/>
      <source name="DataModel.artifacts.attributes"/>
      <target name="Model.Class.Property"/>
   </methodName>
   <methodName name="execute1()"/>
      <source name="DataModel.contextArtifacts.attributes"/>
      <target name="Model.Class.Property"/>
   </methodName>
   <methodName name="execute2()"/>
      <source name="DataModel"/>
      <target name="Model"/>
   </methodName>
</mapping>
```

Figure 13. Mapping file used for translating the INFOTRANS exception and output constraints to transform rules.

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. 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 7. 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 validation rules. □

EXAMPLE 8. 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 13, the rule generator replaces `getArtifacts.*` with artifacts. In general, an access of an element in a collection (e.g., `c.next()`) in π_{code} corresponds to a metamodel element with cardinality 'many' in π_{model}. 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. □

For a library method in a rule, we make no assumptions about any side effects that it may cause. We simply output the library method in the generated rules. If the consumers using the rules (*i.e.,* a validity checker or a transform user or a transform author) know the side-effects of this method, they may choose to keep the rule; otherwise discard it.

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 transform 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.

5. Step 3: Model Validation and Transformation Comprehension

The transform rules inferred in Step 2 of our approach can be used to support model validation and transformation comprehension. Because the rules are stated in the metamodel vocabulary, they can be used in a straight-forward manner to support these tasks.

Our approach distinguishes validation rules from querying rules. In a *validation 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 validation rules are used for checking whether a model is a valid input to a transform, whereas querying rules are used for supporting general transformation comprehension. We illustrate both of these use cases.

5.1 Model Validation

Given a set of validation rules Ψ for a transform $\tau : \mathcal{M}_I \rightarrow \mathcal{M}_O$ and an input model M_I, a validity checker 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 transform can be executed on. However, if at least one of the validation rules is satisfied, M_I is not a valid input to the transform; the transform can fail with an exception when executed on M_I. To check whether a rule $\psi \in \Psi$ is satisfied by M_I, the validity checker 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 validity checker can be used in a batch mode, in which it flags a list of matching rules and corresponding problematic input model elements. Alternatively, the validity checker can be used in an interactive mode; in this mode, while the user is creating an input model, the validity checker flags the problematic input model elements that could cause exceptions. The rules can be translated to a model constraint language such as OCL, for which a validity checker can be constructed by leveraging existing tools or frameworks such as Naomi[14] and EMF Validation Framework in Eclipse.[15]

5.2 Transformation 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.

6. 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.

6.1 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 validation and comprehension tasks.

6.1.1 Experimental setup

We implemented the algorithm shown in Figure 8 using XYLEM. XYLEM uses the WALA analysis infrastructure[16] to construct the call graph and the CFGs. XYLEM performs the

[14] `http://mocl.sourceforge.net/`

[15] `http://help.eclipse.org/galileo/index.jsp?topic=/org.eclipse.emf.validation.doc/tutorials/oclValidationTutorial.html`

[16] `http://wala.sourceforge.net`

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)
Exception Type	Subject	Total	Neg	Positives			Filtered Constraints	Rules	Unique Rules	Rule Predicates			Access-path Len	
				Total	NonSrc	Src				Min	Max	Avg	Min	Max
Null-Pointer exception	Subject-1	1125	1032	193	106	87	79	609	80	1	5	2	2	10
	Subject-2	321	280	41	8	33	19	1008	29	2	6	3.3	3	5
	Subject-3	1301	1222	79	35	44	40	4576	174	1	6	2.6	2	8
	Subject-4	688	639	49	21	28	21	34	23	1	1	1	4	9
Class-cast exception	Subject-1	185	130	55	20	35	11	17	12	1	2	1.2	3	8
	Subject-2	30	27	3	1	2	2	36	2	2	2	2	4	4
	Subject-3	198	180	18	2	16	8	96	37	1	4	1.7	3	6
	Subject-4	71	49	23	16	7	6	12	9	1	1	1	4	8
Array-index exception	Subject-1	12	4	8	6	2	2	2	1	1	1	1	5	5
	Subject-2	2	0	2	0	2	2	36	2	1	1	1	4	4
	Subject-3	10	10	0	0	0	0	0	0	0	0	0	0	0
	Subject-4	3	1	2	2	0	0	0	0	0	0	0	0	0

Table 3. Inferred exception constraints and validation rules.

Subject	Classes	Methods	Bytecode instructions	Time
Subject-1	41	280	4904	285.5s
Subject-2	13	77	1212	15.7s
Subject-3	48	399	5340	280.5s
Subject-4	29	157	2449	33.08s

Table 2. Subjects used in the empirical evaluation.

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. The last column lists the time taken to execute XYLEM on these subjects. These subjects are real model transforms 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` transform a SOMA Service Model [26] to an application-specific ECORE model; these were intermediate models that were eventually transformed to different types of code artifacts. `Subject-2` is the INFOTRANS transform introduced in Section 2.1. `Subject-3` transformed a SOMA Service Model to an RSA Software Services Model.[17]

6.1.2 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 transform authors to examine the computed constraints and identify filters that would remove invalid and bug constraints. We used the filters in the second run of XYLEM. We wrote a simple Java program to translate filtered exception constraints to validation rules and remove duplicate rules. For `Subject-1` and `Subject-2`, the final rule set was examined by the transform authors to determine the validity of the rules. All reported validation rules were found to be valid rules.

6.1.3 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 total number of traversals performed by XYLEM. The maximum number of traversals were performed for null-pointer exceptions. This is expected because dereference statements occur much more frequently in Java programs than typecast statements or statements that access of collections.

Column 4 shows the number of *negatives*—that is, the number of traversals that XYLEM determined could not result in a null-pointer exception, a class-cast exception, or an array-index exception. Column 5 is the number of true positives and is divided into two categories: Column 7 shows the number of positive constraints that were rooted in the source and, therefore, are potential candidates for rules; Column 6 shows the number of positive constraints that were not rooted in the source and, hence, could not be mapped back to the model. For example, `SolutionUtils.specialchars = null` is a "local" predicate that cannot be mapped into the input model and `target.eContainer().getRole()=null` is rooted in the target model and has no mapping into the source model.

Some of the constraints in Column 7 get filtered out during post-processing. Column 8 gives the number of constraints left after the filters have been applied. For example, `source.eContainer().getPackage()=null` gets filtered out since we know that `getPackage()` can never return `null`. For our subjects, we have a set of 30 filters that have been manually specified.

The data illustrate that filters are effective in removing many uninteresting constraints. On average over all subjects

[17] `http://www.ibm.com/developerworks/rational/library/05/510_svc/`

and exceptions, the number of constraints was reduced by over 23%, from 247 initial constraints to 190 constraints, after filtering. The maximum reduction—over 38%—occurred for `Subject-2`.

Column 9 shows the number of validation rules that were translated from the final constraints. As mentioned in Section 4, 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 validation rule. The data in column 9 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 10 illustrates that a very small percentage of the rules were unique rules. 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 11–13, 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 14–15 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.

6.1.4 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 validation and transform comprehension. Moreover, many of the validation 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 widely among our subjects—from 114 to two. The number of path constraints depends on the structure of the program and complexity of the input metamodel; therefore, the variation indicates that our subjects are structured quite differently, in terms of the number of program paths and the input metamodel. The data also demonstrate the effectiveness of XYLEM in that it is able to analyze many paths.

We manually analyzed the code base of INFOTRANS (Subject-2) for each traversal that XYLEM reported to be a negative (Column 4 of Table 3). For null-pointer exceptions, we sampled 210 negatives. Of these, eight were found to be false negatives. Therefore, at eight points in the transform code, exceptions could be thrown because of null values being passed in some input model element, but that were ignored by our analysis. These eight constraints led to two rules, one of which was already computed (by the analysis of a difference dereference point); the other rule was missed by the analysis. Similarly, for class-cast exceptions, we sampled 16 negatives, out of which one was a false negative and would have led to a new rule being identified.

The main source of the false negatives was the presence of calls to external methods for which bytecode was not available for analysis. For such method calls, XYLEM cannot determine whether the return values may be null and hence misses some positives.

Another source of false negatives was failures caused by exceptions that are not analyzed by our implementation. As discussed in Section 3.6, exceptions thrown by calls to external methods cause the analysis to be incomplete. An example of such a call that we found is

```
new Integer(source.getMultiplicity()).intValue()
```

If the string referenced by `source.getMultiplicity()` were not a parsable integer, a `NumberFormatException` would be thrown. Because our current implementation handles only a limited set of exceptions, it cannot compute constraints for such statements.

6.2 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.

6.2.1 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 transformtion 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 transform code to identify violated input model constraints. Based on these factors, we grouped the participants into three categories: expert (one participant, referred to as

Participant	Task T_1: failing execution		Task T_2: incomplete output	
	$T_{(1,wr)}$	$T_{(1,r)}$	$T_{(2,wr)}$	$T_{(2,r)}$
E1	5	3 (60%)	2	1 (50%)
I1	7	4 (57%)	14	2 (14%)
I2	6	4 (67%)	8	3 (38%)
I3	13	5 (38%)	7	6 (86%)
N1	16	7 (41%)	14	7 (50%)

Table 4. Time taken by the participants to complete the tasks.

E1), intermediate (three participants, referred to as I1, I2, and I3), 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 subtasks, one in which the participants had to debug the problem without using the transform rules (T_{wr}), and another in which the participants had to debug the problem while guided by the 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 each task, the participants were asked to fix the input models. For $T_{(1,wr)}$ and $T_{(2,wr)}$, the participants were given access to the transform code and were also allowed to use code-debugging features. For the $T_{(1,r)}$ and $T_{(2,r)}$, the participants were allowed to use the rules only (with no access to the transform code). Thus, we simulated the scenario in which transform users have to debug their models without needing to examine the transform source code. The transform rules were created by running XYLEM on the INFOTRANS transform; the computed rules were augmented with manually created querying rules for output constraints.

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

6.2.2 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 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 (from 14 minutes to two minutes) occurred for user I1 for task T_2. The minimum reduction occurred for user I3 for task T_2.

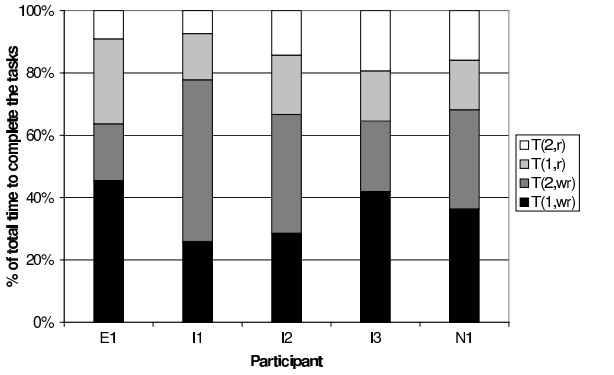

Figure 14. Percentage of total time spent by the participants on the four tasks.

Figure 14 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 the models without the rules, whereas they spent significantly less time (22% to 38%) in fixing the 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 transforms was different from debugging normal Java applications, as the inputs to transforms 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 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 could return the matching input model elements for the rules that explain a failing transformation. 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.

6.2.3 Discussion

Although our study is limited in nature, the results support 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.

7. Related Work

There exists a rich body of work in the area of verification and validation of model transformations. However, all of the existing research focuses on checking the correctness of transforms.

Giese et al. [12] present an approach, based on formal specifications and theorem proving, for verifying the

correctness of a model-to-code transformation algorithm. Narayanan and Karsai [23] present a verification technique that uses bisimulation to check whether the semantic properties of the input in a particular execution of a transform are preserved in the output for that execution. Their approach focuses on transform implementations that are based on graph transformations. Lano and Clark [18] present a constraint-based technique for specifying and verifying transforms. Unlike these approaches, our work does not focus on transform verification. Instead, the goal of our work is to assist users in creating valid input models to a transform and in identifying problems with the input model for a failing or incomplete transformation.

In addition to verification techniques, many researchers have addressed problems that MDD poses for testing activities. Baudry et al. [1] present of overview of MDD characteristics that can complicate different testing tasks. For example, the complexity of input models can complicate test-input generation, and the heterogeneity of transform implementations can make definition of test adequacy difficult. Existing research has addressed many such testing problems, such as test-input generation (*e.g.,* [3, 7]), test-oracle construction (*e.g.,* [20]), definition of test-adequacy criteria (*e.g.,* [9, 11]), assessment of test quality (*e.g.,* [19]), and definition of fault models (*e.g.,* [16]). Our work addresses an important testing-related task—debugging of failing and incomplete transformations—that has largely been ignored; therefore, it fills a gap in existing research.

Our analysis for computing constraints is similar to the computation of weakest preconditions (*e.g.,* [6, 8]). However, we apply the analysis to the domain of MDD, in which the inferred constraints are mapped to rules that are stated in the language of the input metamodel. Existing research has not explored this application of precondition analysis.

Analysis for identifying input constraints has most commonly been used for generating test inputs. Compared with such test-data generation techniques that use symbolic execution to generate test inputs (*e.g.,* [13, 14, 25]), our approach does not generate test inputs. Therefore, its effectiveness is not dependent on the power of constraint solvers, which despite recent advances, continue to have practical limitations. For our application, the constraints are mapped to model-level rules.

Buse and Weimer [4] present a static analysis for identifying exception conditions to assist with documentation. Their approach locates exception-throwing statements and symbolically tracks paths to those statements. The symbolic execution generates predicates describing feasible paths, and yields a boolean formula over program variables. This formula is used to generate human-readable documentation. In contrast, our approach computes exception and output constraints that are mapped to model-level rules and used for model validation and transformation comprehension.

8. Summary and Future Work

In this paper, we presented an approach for assisting users of model transforms in debugging their input models without examining the transform code. The approach uses static code analysis to compute constraints on the input model under which a transform 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 validation and comprehension rules that are stated in the metamodel language. The rules are used to support model validation and transformation comprehension: the validation rules can be used for checking whether a metamodel instance is a valid input to a transform, whereas the comprehension 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 transform 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 transforms In this paper, we focused on model-to-model transforms that are written in Java. However, models are often represented using XML and XSLT is frequently used in practice for writing transforms. Thus, future research could extend our approach to handle transforms implemented in XSLT (or, other transform-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. Reference [15] presents an interesting interactive debugging approach, in which 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 transform code is not available for inspection. Future improvements could also combine static analysis with dynamic information gathered from transform executions to provide more accurate results to users.

References

[1] B. Baudry, S. Ghosh, F. Fleurey, R. France, Y. Le Traon, and J.-M. Mottu. Barriers to systematic model transformation testing. *Communications of the ACM*, 2009. To appear.

[2] I. D. Baxter. Design maintenance systems. *Communications of the ACM*, 35(4):73–89, April 1992.

[3] E. Brottier, F. Fleurey, B. Baudry, and Y. Le Traon. Metamodel-based test generation for model transformations: An algorithm and a tool. In *Proc. of the 17th Intl. Symp.on Softw. Reliability Eng.*, pages 85–94, November 2006.

[4] R. P. L. Buse and W. R. Weimer. Automatic documentation inference for exceptions. In *Proc. of the Intl. Symp. on Softw. Testing and Analysis*, pages 273–281, July 2008.

[5] J. Cabot and R. Clarisó. UML/OCL verification in practice. In *Proc. of the 1st Intl. Workshop on Challenges in Model-Driven Softw. Eng.*, pages 31–35, September 2008.

[6] S. Chandra, S. J. Fink, and M. Sridharan. Snugglebug: A powerful approach to weakest preconditions. In *Proc. of the ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 363–374, June 2009.

[7] T. T. Dinh-Trong, S. Ghosh, and R. B. France. A systematic approach to generate inputs to test UML design models. In *Proc. of the 17th Intl. Symp.on Softw. Reliability Eng.*, pages 95–104, November 2006.

[8] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. of the ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 234–245, June 2002.

[9] F. Fleurey, J. Steel, and B. Baudry. Validation in model-driven engineering: Testing model transformations. In *Proc. of the 1st Intl. Workshop on Model, Design and Validation*, pages 29–40, November 2004.

[10] D. S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley and Sons, 2003.

[11] S. Ghosh, R. France, C. Braganza, N. Kawane, A. Andrews, and O. Pilskalns. Test adequacy assessment for UML design model testing. In *Proc. of the 14th Intl. Symp.on Softw. Reliability Eng.*, pages 332–346, November 2003.

[12] H. Giese, S. Glesner, J. Leitner, W. Schäfer, and R. Wagner. Towards verified model transformations. In *Proc. of the 3rd Workshop on Model Design and Validation*, pages 78–93, October 2006.

[13] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proc. of the ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 213–223, June 2005.

[14] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[15] A. J. Ko and B. A. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *Proc. of the 30th Intl. Conf. on Softw. Eng.*, pages 301–310, May 2008.

[16] Jochen M. Küster and Mohamed Abd el razik. Validation of model transformations - First experiences using a white box approach. In *Proc. of the 3rd Workshop on Model Design and Validation*, pages 62–77, October 2006.

[17] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proc. of the ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 235–248, June 1992.

[18] K. Lano and D. Clark. Model transformation specification and verification. In *Proc. of the 8th Intl. Conf. on Quality Softw.*, pages 45–54, August 2008.

[19] J.-M. Mottu, B. Baudry, and Y. Le Traon. Mutation analysis testing for model transformations. In *Proc. of the Model Driven Architecture – Foundations and Applications, 2nd European Conf.*, volume 4066 of *Lecture Notes in Computer Science*, pages 396–390, 2006.

[20] J.-M. Mottu, B. Baudry, and Y. Le Traon. Reusable MDA components: A testing-for-trust approach. In *Proc. of the 9th Intl. Conf. on Model Driven Eng. Lang. and Syst.*, volume 4199 of *Lecture Notes in Computer Science*, pages 589–603, 2006.

[21] M. Nanda, C. Grothoff, and S. Chandra. Deriving object typestates in the presence of inter-object references. In *Proc. of the 20th ACM SIGPLAN Conf. on Object-Oriented Prog., Syst., Lang., and Applications*, pages 77–96, October 2005.

[22] M. G. Nanda and S. Sinha. Accurate interprocedural null-dereference analysis for Java. In *Proc. of the 31st Intl. Conf. on Softw. Eng.*, pages 133–143, May 2009.

[23] A. Narayanan and G. Karsai. Towards verifying model transformations. *Electron. Notes Theor. Comput. Sci.*, 211:191–200, 2008.

[24] D. C. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2):25–31, February 2006.

[25] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proc. of the Intl. Symp. on Softw. Testing and Analysis*, pages 97–107, July 2004.

[26] L. J. Zhang, N. Zhou, Y. M. Chee, A. Jalaldeen, K. Ponnalagu, R. R. Sindhgatta, A. Arsanjani, and F. Bernardini. SOMA-ME: A platform for the model-driven design of SOA solutions. *IBM Systems Journal*, 47(3):397–413, 2008.