Symbolic Execution of High-Level Transformations [*]

Ahmad Salim Al-Sibahi
IT University of Copenhagen, Denmark
asal@itu.dk

Aleksandar S. Dimovski
IT University of Copenhagen, Denmark
adim@itu.dk

Andrzej Wąsowski
IT University of Copenhagen, Denmark
wasowski@itu.dk

Abstract

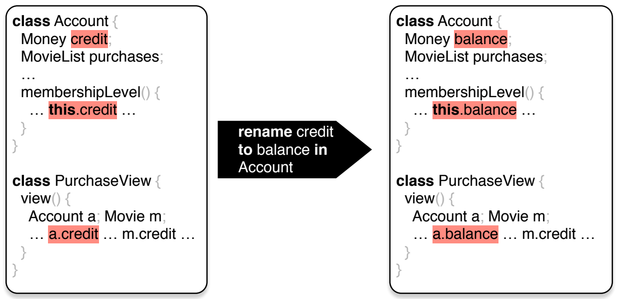
Transformations form an important part of developing domain specific languages, where they are used to provide semantics for typing and evaluation. Yet, few solutions exist for verifying transformations written in expressive high-level transformation languages. We take a step towards that goal, by developing a general symbolic execution technique that handles programs written in these high-level transformation languages. We use logical constraints to describe structured symbolic values, including containment, acyclicity, simple unordered collections (sets) and to handle deep type-based querying of syntax hierarchies. We evaluate this symbolic execution technique on a collection of refactoring and model transformation programs, showing that the white-box test generation tool based on symbolic execution obtains better code coverage than a black box test generator for such programs in almost all tested cases.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Symbolic execution; D.3.2 [*Language Classifications*]: Very high-level languages; I.2.2 [*Automatic Programming*]: Program transformation, Program verification

Keywords program transformation, model transformation, symbolic execution, automated white-box test generation

1. Introduction

Transformations are everywhere: from being used to prettily display structured data available in JSON or XML formats in many websites, to forming the core of language workbenches such as Spoofax (Kats and Visser 2010), where they provide name resolution, typing and dynamic semantics for Domain Specific Languages (DSLs). Consider the *rename field* refactoring in Fig. 1 as an example of a transformation. It changes the name of a target field in the definition of the target class and ensures that all relevant field accesses use the new field name (Fowler 1999).



```
class Account {
  Money credit;
  MovieList purchases;
  ...
  membershipLevel() {
    ... this.credit ...
  }
}

class PurchaseView {
  view() {
    Account a; Movie m;
    ... a.credit ... m.credit ...
  }
}
```

```
class Account {
  Money balance;
  MovieList purchases;
  ...
  membershipLevel() {
    ... this.balance ...
  }
}

class PurchaseView {
  view() {
    Account a; Movie m;
    ... a.balance ... m.credit ...
  }
}
```

Figure 1: The *Rename-Field* refactoring: rename the definition of credit to balance and update all references accordingly.

While one could write transformations like *rename field* in C or Java, they are optimally written in a specialized transformation language or framework, proposed across various communities such as the programming language community (TXL (Cordy 2006), Stratego (Bravenboer et al. 2008), Uniplate (Mitchell and Runciman 2007), Kiama (Sloane 2011)), the model transformation community (ATL (Jouault and Kurtev 2005), Epsilon (Kolovos et al. 2008), QVT (Object Management Group 2011)) and the concurrency theory community (Maude (Troya and Vallecillo 2011)). All these languages support some form of *type-directed querying and manipulation*. Type-directed querying allows matching structural patterns in structured data by following types of objects and references between them, while type-directed manipulation allows rewriting patterns to structures with new types and new references. Type-directed querying and manipulation is *deep*, unlike in classic functional programming, so target patterns can be matched anywhere in a syntax tree. In our example in Fig. 1, a type-directed query makes it possible to retrieve all accesses to the credit field where the target expression has type Account, using only a couple of lines.

[*] Partially funded by Danish Council for Independent Research, grant no. 0602-02327B

SLE'16, October 31 – November 1, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4447-0/16/10...
http://dx.doi.org/10.1145/2997364.2997382

Transformations are complex programs and as such prone to bugs; for our *rename field* example, a bug could be that the new field name clashes with an existing one in the class. Due to the complexity of transformations these bugs are hard to find and expensive to fix, yet transformations form the core of daily-used language implementations and tools. Developing formal techniques and automated tools to verify the correctness of these transformations is therefore important to increase the trustworthiness of our language implementations and tools (Cadar and Donaldson 2016; Schäfer et al. 2009; Hoare 2005).

We aim to take a step towards achieving that goal by presenting a foundational symbolic execution technique for high-level transformation languages. Our technique handles target high-level transformation features—containment, set expressions, type-directed querying and manipulation, and fixedpoint iteration—as first-class to make it feasible to use in practice. Concretely, our contributions are:

- TRON, a compact formally defined imperative language suitable for theoretical development of analysis methods for transformations, including type-directed querying and manipulation; the language has been designed to capture key properties in this space.
- A formal symbolic execution technique for TRON that deals with complex concepts such as symbolic sets, ownership constraints and deep type-directed operations.
- An evaluation of the symbolic execution technique when used for white-box test generation using realistic model transformations and refactorings, showing that our symbolic executor makes effective white-box test generation for transformation feasible.
- A comparison of our symbolic execution technique to object-oriented symbolic executors, highlighting the difficulties of dealing with target high-level features as second-class.

Our intended audience are researchers in programming languages and software engineering, who recognize the need of first-class analysis techniques for transformations. By providing a useful symbolic semantics, we hope that this work can influence efforts in building tools for test generation, static analysis and verification of such transformations.

2. Overview

Running Example We start by discussing the example in Fig. 1 in more detail. Observe that the refactoring program needs two important parts: the *type definitions* (sometimes called *meta-model*) for the data and the actual transformation *code*. We will use (minimalistic) class diagrams to show the former, and TRON, our compact formally defined transformation language, to show the latter. These two notational choices incorporate some key common characteristics of transformations, which we discuss below.

Fig. 2a shows the types for the abstract syntax of a hypothetical object-oriented language. We show the classes[1] and properties relevant for our refactoring, while omitting irrelevant details. In our example, each class has a name (an attribute), and *contains* two collections, one for fields and one for methods. Recall that in class diagrams, a black diamond is used to decorate *containment* references. Containment is traditionally found in object-oriented modeling, but also exists in algebraic data types of functional programming languages, in grammar-based languages like TXL, and in XML documents.

Additionally, each class may also simply *refer* to a possible super-class. Note that the simple *reference* is denoted using a line without the diamond symbol. The mixture of classes, with containment references, simple references, and attributes of simple types is typical of transformation languages, so we include these constructs in TRON.

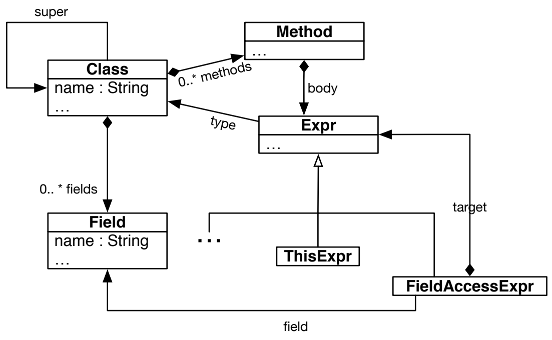
In the example, each method has a body which we—for simplicity of presentation—allow only to be expressions. Expressions themselves can come in many different kinds (thus the use of *inheritance*), but we only show expressions representing 'this' and field access expressions since they are the ones relevant for the example.

A simplified implementation of the *rename field* refactoring, is shown in Fig. 2b. The implementation of the refactoring is presented in TRON. We introduce TRON and its semantics in Sect. 3–4, but let us discuss the example based on general intuitions. In the top we list the input parameters (references to a class, the field with the old name, and the replacing field with the new name) and the application precondition (the old field has to be contained in the class's fields, whereas the new must not).

We begin the refactoring by removing the old field definition from the fields of the class and adding the new field definition (Line 5). Then, in Line 6, all field access expressions in the class are matched and gathered into a single set using a *deep type-directed query*, which collects instances of FieldAccessExpr contained transitively in the input class. A language without first class support for such kind of queries usually requires implementing traversal algorithms for the structure in question (e.g., in the form of visitors), or use of dynamic dispatch, reflection or other type-access mechanism to select the right nodes. This capability is however available directly in high-level transformations languages, such as those mentioned in Sect. 1, and therefore is included in TRON.

After the deep type-directed query, Line 6 binds each element of the matched objects to *faexpr* executing Lines 7–9 for each of these objects. If the expression accesses

[1] The abstract syntax types do not describe the syntax of TRON itself. They are a description *in* TRON of data manipulated by our running example, which is a TRON program, whose input data happens to represent hypothetical object-oriented programs. In particular, do not confuse TRON (meta) classes as types of objects, with the objects they type which are classes in the subject input programs that are refactored in the example.



(a) Abstract syntax for simple object-oriented programs

```
1 input:  target_class: Class, old_field: Field, new_field: Field
2 precondition: old_field ∈ target_class.fields
3                  ∧ new_field ∉ target_class.fields
4
5 // the refactoring program
6 target_class.fields := (target_class.fields \ old_field) ∪ new_field
7 foreach faexpr ∈ target_class match* FieldAccessExpr do
8   if faexpr.field = old_field ∧
9       faexpr.target.type = target_class then
10         faexpr.field := new_field
11   else skip
```

(b) A Refactoring in TRON

Figure 2: A simplified version of the *rename-field* refactoring example in TRON

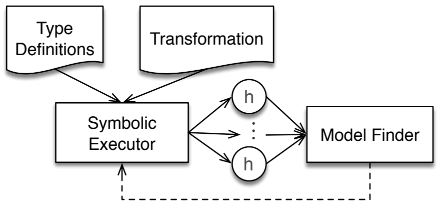


Figure 3: High level architecture of the symbolic executor.

the refactored field (Line 7-8) then the field reference is updated to point at the new field (Line 9). It is typical for the transformation languages that references are redirected or attributes are changed. This happens either imperatively using destructive updates (like in the example), or in a pure way by copying, and usually continues until no more changes are possible. TRON has the `foreach` statement (and also a simpler `fix` statement) to emulate the fixed point semantics of these languages. We chose to make TRON imperative so we can reason about destructive updates, which are allowed in many transformation languages like ATL and Kiama.

Symbolic Execution of Transformations An effective way to check whether there is any bug present in transformations—like *rename field* refactoring presented above—is by using symbolic execution, which is able to systematically explore the various program paths. An overview of our symbolic execution technique for transformations is presented in Fig. 3. The symbolic executor expects a transformation written in TRON as an input, along with the required type definitions. The initial step is to run the symbolic executor (see Sect. 4) on the input transformation and generate a finite set of path conditions. These path conditions are logical formulæ constraining the shape, types and range of input data, achieved by refining input constraints according to the semantics of each statement in the given transformation.

Intermediately, we use the model finder to prune those paths which produce unsatisfiable formulæ so that only valid paths are considered. In our implementation, the model finder uses the relational constraint solver KodKod (Torlak and Jackson 2007) to check the existence of a suitable model satisfying a target formula within a bounded scope, possibly failing when either the formula is unsatisfiable or the scope is too small.

3. A Demonstration Language

TRON is a compact theoretical transformation language, incorporating characteristic features of high-level constructs of languages discussed in Sect. 2; Tbl. 1 shows how these features are captured in TRON. TRON is a decoy language, so that the core of our ideas remain applicable to real-world transformation languages. We developed it as a methodological device, to keep the formal work, discussions, and the presentation focused, and to allow agile experimentation; TRON is *not* meant to be used by programmers.

We present TRON in two parts: a) the meta-model that captures structures of the manipulated data and b) the operational part of the language that describes computations.

Notation. We use r^* to denote the reflexive-transitive closure of a binary relation r. We use $\wp(A)$ to denote the power set of A. For a particular function $f \in A \to B$, we use graph f to represent the set $\{\langle x, f(x)\rangle | x \in \text{dom } f\}$. We use $f[a \mapsto b]$ to represent function updates, so that $f[a \mapsto b](a) = b$ and $f[a \mapsto b](a') = f(a')$ when $a' \neq a$.

Data Model. The data in TRON is described by types that capture the common features of rewriting languages: constructors, containment, references and generalization. It is essentially a formal model for the kind of structures like the one represented in Fig. 2a.

A data model is a tuple: $\langle \text{Class}, \text{Field}, \mathsf{gen}, \mathsf{ref} \rangle$, where Class is the set of *classes*, Field is the set of *fields*, partitioned into *contained* fields $\text{Field}_{\blacklozenge}$ and *referenced* fields $\text{Field}_{\rightsquigarrow}$. Later, we use c to range over class names (Class), and f to range over field names (Field). A class has at most one superclass, described by the generalization relation: $\mathsf{gen} \subseteq \text{Class} \times \text{Class}$, where $c \;\mathsf{gen}\; c'$ means that c is a subtype of c'. Each field has a corresponding type, a class. This is represented by the references relation $\mathrm{ref} \subseteq \text{Class} \times \text{Field} \times \text{Class}$, where $\mathsf{ref}(c, f, c')$ means that the class c has a field f of type c'. We generally expect that gen has the expected properties of a generalization relation, namely

Feature \ Language	ATL	Scala	Haskell	Maude
Containment	Containment references	Case Classes	Algebraic Data Types	Many-Sorted Terms
Set expressions	OCL collections and collection operations	Standard library	Standard library	Standard library
Shallow matching	Type testing via `oclIsKindOf`	Pattern matching	Pattern matching	Rewrite rules
Deep matching	Transformation rule definition	Rewrite rules and strategies via Kiama	Generic traversal via Uniplate	Rewrite rules and strategies
Fixedpoint iteration	Lazy rules, recursive helpers	Recursive functions, Kiama strategies	Recursive functions	Rewrite strategies

Table 1: Relating TRON features to existing high-level transformation languages

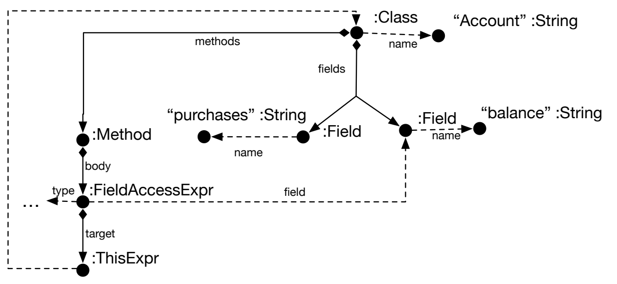


Figure 4: A heap instantiating one model of Fig. 2a, inspired by the Account class in Fig. 1. Dots (●) represent instances, diamond affixed lines represent containment links, dashed lines represent simple links.

that there is a strict ordering of generalization (no cycles); similarly, we expect that reference definitions in ref are not overriden by subtypes, i.e. if for any class c a supertype has defined a typing for a field f, then c must have the same typing for f.

We will let $\mathsf{fields}(c)$ be the function that gets all fields defined for a class c or any of its supertypes, and is defined as follows: $\mathsf{fields}(c) = \{\langle f, c''\rangle | c \,\mathsf{gen}^*\, c' \wedge \mathsf{ref}(c', f, c'')\}$. We do not explicitly handle simple types in our formal model. Simple types can be modeled using classes from a theoretical point of view. For instance we can assume a class Integer with instances representing integer numbers. Then integer attributes can be modeled as references to this class. We do handle simple types in our symbolic executor, in the same way as other symbolic executors do, using symbolic variables of corresponding simple types.

Heap Representation. Concrete TRON programs are executed over finite concrete heaps ($h \in \text{Instance} \times \text{Field} \to \wp\,(\text{Instance})$) that contain instances organized into structures using containment links and simple links (a link is a concrete instantiation of a field). In particular each link f of an instance o, can point to a set of instances $\mathbb{O}$. Instances are typed at runtime using a type environment ($\Gamma \in \text{Instance} \to \text{Class}$). An example heap is shown in Fig. 4, which describes a possible definition of the Account class from Sect. 1.

For the remainder of this paper we will only consider well-formed heaps where all instances are typed and their structure conforms to the static typing provided by the data model. Furthermore, we assume that in well-formed heaps each instance can at most be pointed to by a single containment link (no sharing) and that there are no cycles in containment (acyclicity). Note that these restrictions do not apply for simple links, which still allow cycles and sharing.

Abstract Syntax. The core TRON constructs include access to variables and fields, constants, object construction, assignment, sequencing and branching. The syntax is summarized in the following grammar:

$$
\begin{aligned}
\overline{\text{SetExpr}} \ni \overline{e} &::= x \mid \emptyset \mid \overline{e}_1 \cup \overline{e}_2 \mid \overline{e}_1 \cap \overline{e}_2 \mid \overline{e}_1 \setminus \overline{e}_2\\
\overline{\text{BoolExpr}} \ni \overline{b} &::= \overline{e}_1 \subseteq \overline{e}_2 \mid \overline{e}_1 = \overline{e}_2 \mid \neg \overline{b} \mid \overline{b}_1 \wedge \overline{b}_2\\
\overline{\text{MatchExpr}} \ni \overline{me} &::= \overline{e} \mid \overline{e} \ \mathsf{match}\ c \mid \overline{e}\ \mathsf{match}^*\ c\\
\text{Statement} \ni s &::= \mathsf{skip} \mid s_1; s_2 \,|\, x := \overline{e} \mid x := \overline{e}.f\\
&\quad|\, x := \mathsf{new}\ c \,|\, \overline{e}_1.f := \overline{e}_2 \,|\, \mathsf{if}\ \overline{b}\, \mathsf{then}\ s_1\ \mathsf{else}\ s_2\\
&\quad|\, \mathsf{foreach}\ x\ \mathsf{in}\ \overline{me}\ \mathsf{do}\ s \,|\, \mathsf{fix}\,\overline{e}\ \mathsf{do}\ s
\end{aligned}
$$

where x is a variable, f is a field name, and c is a class name. The set expressions $\overline{e}$ and Boolean expressions $\overline{b}$ are standard. Match expressions ($\overline{me}$) include "$\overline{e}$ `match` c" which allows finding all objects computed by $\overline{e}$ that are instances of class c. For example, given a set of expressions $exprs = \{te_1, te_2, fae_1, fae_2\}$ where te_i is of type ThisExpr (from Fig. 2a) and fae_i is of type FieldAccessExpr, then the expression "*exprs* `match` ThisExpr" would return the set $\{te_1, te_2\}$, similarly "*exprs* `match` FieldAccessExpr" returns $\{fae_1, fae_2\}$ and "*exprs* `match` Expr" return the complete set *exprs*. A deep variant of the pattern matching, $\overline{e}$ `match`* c, is also provided. It matches objects nested at an arbitrary depth inside other objects, following the containment references ($\mathsf{ref}_{\blacklozenge}$). This is similar to the matching capabilities in many of the model transformation, term and graph rewriting languages. A classical example here would be to get all variables in a term, i.e., the expression *expr* `match`* Var—for a class Var representing variables—would return a set that has all variables transitively contained in *expr*.

Most of the statements, s, are standard formulations from Java or `IMP`; from left to right, the statements are: skip, sequencing, branching, variable assignment, assignment of a field value, object creation (new) and assignment to a field.

There are two looping constructs in TRON. The "foreach x in $\overline{me}$ do s" iterates over the set of elements matched by $\overline{me}$, binding each element to x, and executes then statement s for each of them. The "fix $\overline{e}$ do s" loop executes the body s, and continues to do so as long as the values of $\overline{e}$ after and before iteration differ; therefore expression $\overline{e}$ defines the part of the heap which is relevant for this fixed point iteration (a control condition). By allowing the statement to explicitly depend on a local control condition, it is possible to create temporary helper values on the heap (outside $\overline{e}$) without influencing the loop termination. This allows explicit modeling of the implicit fix point iteration that is also supported by many high-level transformation languages where rewrite rules are repeatedly applied until no rule is further applicable.

4. Symbolic Execution

We discuss the main design principles of our symbolic executor. Although, the technique has been developed for TRON, the design decisions were driven by the desire to handle the language features incorporated by TRON, which are selected from several transformation languages.

4.1 Symbolically Representing Rich States

Symbolic execution uses *symbols* (fresh variables) to represent unknown values of (King 1976), which we denote with small Latin letters followed by a question mark ($x^?$). We present below the representation of state our symbolic executor maintains to correctly constrain the legal shapes of possible concrete stores and heaps on each program path.

Spatial Constraints. Since transformations manipulate structured data, not just simple values, the symbolic states of our executor describe primarily the possible shapes of the memory heap. Following other symbolic executors for object-oriented languages (Khurshid et al. 2003), we use *spatial constraints* to restrict the shapes admitted by an execution path. These constraints are first order formulae restricting values that are pointed to by links. In the style of the Lazier# algorithm (Deng et al. 2012), we distinguish between two kinds of symbolic objects: *symbolic instances* and *symbolic references*.

A *symbolic instance* ($o \in \mathrm{Instance}$) abstracts over a unique instance. Instances cannot alias, so two different symbolic instances always point to two different class instances in memory, even if they have the same type. A *symbolic reference* ($x^?, y^? \in \mathrm{Symbol}$) points to a class instance that may be aliased by other reference symbols, and, indeed, by some symbolic instances. The separation of symbolic instances and symbolic references allows to separate reasoning about the structure of the representation of the data from aliasing by references. We can lazily reason about aliasing without committing pre-maturely to a particular concretization of the heap structure. This is particularly important for our symbolic executor, as it handles deep containment constraints, which are hard to reason about and are heavily affected by aliasing (more about deep containment constraints below).

In traditional symbolic execution (Khurshid et al. 2003), whenever a field is accessed, the executor branches to initialize it to a new symbolic instance, or to alias an existing symbolic instance. In contrast, the Lazier# algorithm, simply assigns a distinct symbolic reference to each fresh field access and aliasing is only explicitly treated if the substructure of that symbolic reference is further explored.

For objects created using new, we eagerly generate a new concrete instance and exclude it from aliasing with pre-existing symbolic references, as new objects cannot alias previously existing ones (assuming correctness of the memory manager). To emphasize this in the rules below, we mark the explicitly created instances with a dagger ($o^\dagger$).

Set Symbols. In addition to ordinary symbolic references, we introduce *symbolic reference sets*, or *set symbols* for short ($X^?, Y^? \in \mathrm{SetSymbol}$). These symbols abstract over finite sets of instances with unknown cardinality. This addition may seem very simple at first, but is key for our symbolic executor: it allows us to range over sets without prematurely concretizing their cardinality, contained objects, or their structure and aliasing.

Set Expressions and Set Constraints. Set symbols can be combined using symbolic set expressions:

$$\mathrm{SetExpr} \ni e \ ::= \ X^? \mid \emptyset \mid \{x_0^?, \ldots, x_n^?\} \mid e_1 \cup e_2 \mid e_1 \cap e_2 \mid e_1 \setminus e_2$$

The symbolic set expressions mimic the set expressions of TRON, presented in Sect. 3, but without match expressions and with support for literal set constructors over simple symbolic references $\{x_1^?, \ldots, x_n^?\}$. The meaning of the latter is a set of a fixed cardinality n, whose all elements are distinct (so, as a side effect, it also precludes aliasing between symbols listed). We use it to concretize the cardinality and content of sets during iteration.

Set expressions are embedded into constraints in a standard manner, using subset and equality constraints:

$$\mathrm{BoolExpr} \ni b \ ::= e_1 \subseteq e_2 \mid e_1 = e_2 \mid \neg b \mid b_1 \wedge b_2$$

During symbolic execution the set comprehensions and reference symbols interplay to our benefit, allowing to describe assumptions about sets more lazily. For example, consider the constraint that equates two sets of cardinality 3 of unknown references: $\{x_1^?, x_2^?, x_3^?\} = \{y_1^?, y_2^?, y_3^?\}$. Generating this constraint allows to avoid deciding prematurely, which of the six possible aliasing configurations between x_is and y_js we are seeing, something which would not scale if done repeatedly.

Containment Constraints. A special feature of our symbolic executor is its ability to reason about the deep containment constraints of the manipulated data structures, which are extremely common in language processing (abstract syntax trees) and in data modeling. Besides eliminating many false positives, reasoning about containment also allows implementing deep matching.

To model deep containment constraints, we define a containment relation as the union of all links typed by containment fields, and insist that, for any two objects, their containments sets (the transitive closure of the containment relation) are disjoint. Furthermore, we enforce the acyclicity of the containment relation, ensuring that the irreflexive transitive closure of containment does not contain the identity pair for any object. We are using a solver (KodKod) that allows reasoning about transitive closures.

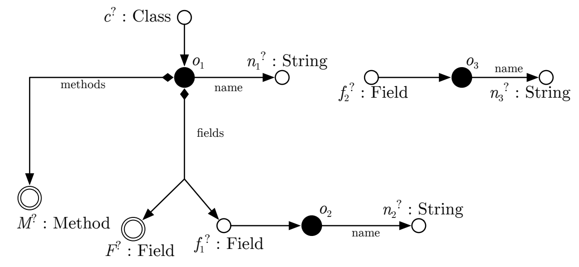
In order to perform symbolic deep matching, we on-demand bind a symbolic *set of containments* to each instance o used in a deep-matching query against a type c. Descendants are instances of c reachable by containment links from o. The set is constrained during execution to contain any referenced containments of o that have been given symbolic names.

Type Constraints. We introduce type approximation in our symbolic executor, in order to not concretize types of instances (objects) prematurely. Many transformation rules operate on data constrained by types with inheritance, so the actual type of parameters might be unknown during symbolic execution. We maintain a bounding constraint on types, and refine it during execution by-need during branching and concretization cycles.

A type constraint environment (Γ) maps each symbolic reference, set symbol, and each, symbolic instance o to a type bound $\langle cs_{\text{in}}, cs_{\text{ex}}\rangle$. The bound restricts the types of concrete values assignable to a symbol in question. A type bound is a tuple $\langle cs_{\text{in}}, cs_{\text{ex}}\rangle$ where the first component cs_{in} is a set of classes that specify the possible supertypes of a symbol and the second component cs_{ex} is a set of classes that specify excluded supertypes of a symbol, e.g. $\langle\{\text{Expr}\}, \{\text{ThisExpr}, \text{FieldAccessExpr}\}\rangle$ represents all expressions (subtypes of Expr) that are not 'this' and field access expressions (not subtypes of ThisExpr or FieldAccessExpr).

We only consider type bounds $\langle cs_{\text{in}}, cs_{\text{ex}}\rangle$ that are *well-formed*. That is: (i) the set of possible super-types cs_{in} cannot be empty, and (ii) none of the super-types is excluded: there is no class $c \in cs_{\text{in}}$ which is a subtype of an excluded supertype $c' \in cs_{\text{ex}}$. We also maintain an invariant that the set of possible super-types cs_{in} given by Γ is a singleton for symbolic instances. We will simply write c as a shorthand for $\langle\{c\}, \{c' \mid c' \text{ gen } c\}\rangle$ when the type of an element is precisely known (basically a type constraint stating that an object's type is a subtype of c but not of any of its subtypes).

Symbolic Heaps. A *symbolic heap* combines all types of constraints discussed above to describe possible concrete heaps and typings that could have been created during the execution. We define a symbolic heap to be a tuple $\langle z, \ell, d, \Gamma, b\rangle$, where $z \in \text{Symbol} \to \text{Instance}$ is a symbolic reference environment—partial mapping of symbolic references to



Symbolic references $z = [c^? \mapsto o_1, f_1^? \mapsto o_2, f_2^? \mapsto o_3]$

Symbolic instances
$\ell = [\,\langle o_1, \text{name}\rangle \mapsto n_1^?, \langle o_1, \text{methods}\rangle \mapsto M^?, \langle o_1, \text{fields}\rangle \mapsto F^? \uplus \{f_1^?\}, \langle o_2, \text{name}\rangle \mapsto n_2^?, \langle o_3, \text{name}\rangle \mapsto n_3^?]$

Containment constraints $d = []$ (not accumulated yet)

Type constraints $\Gamma = [o_1 \mapsto \text{Class}, o_2 \mapsto \text{Field}, o_3 \mapsto \text{Field}, c^? \mapsto \text{Class}, n_1^? \mapsto \text{String}, f_1^? \mapsto \text{Field}, n_2^? \mapsto \text{String}, f_2^? \mapsto \text{Field}, n_3^? \mapsto \text{String}, M^? \mapsto \text{Method}, F^? \mapsto \text{Field}]$

Path condition $b = \mathbf{true}$ (not accumulated yet)

Figure 5: An example heap for an initial state of an execution

symbolic instances that they are constrained to point to; $\ell \in \text{Instance} \times \text{Field} \to \text{SetExpr}$ collects the symbolic instances, by mapping fields of symbolic instances to symbolic set expressions; d is an environment storing deep containment constraints $d \in \text{Instance} \times \text{Class} \to \text{SetExpr}$ for all symbolic instances, Γ is the type constraint environment, and b is the path constraint so far, in the execution leading to this symbolic heap.

An example symbolic heap is presented in Fig. 5 using both the above syntax and a diagram[2]. Dot vertices (●) denote symbolic instances, white circles (○) denote symbolic references and large double-stroked white circles (◎) denote symbolic reference sets. We say that a symbolic heap h is *satisfiable* if there exists a concrete heap and a concrete typing environment that satisfy all the constraints of h[3] . One symbolic heap *is stronger* than the other if all models (here all satisfying concrete heaps) of the former are also models of the latter. For conciseness, we let $\langle z, \ell, d, \Gamma, b_1\rangle \wedge b_2$ mean $\langle z, \ell, d, \Gamma, b_1 \wedge b_2\rangle$.

4.2 Manipulating Symbolic State During Execution

Fig. 6 shows an example path of the symbolic executor when executing the Rename-Field refactoring from Fig. 2b starting with the symbolic state presented in Fig. 5. The execution proceeds in the following steps:

- The initial statement on line 5 replaces the old field (represented by symbol $f_1^?$) with the new field ($f_2^?$), such that the 'fields' reference of the target class ($c^?$) now points at $f_2^?$ instead of $f_1^?$.

[2] We use our own diagram notation for objects instead of the UML one because it is more compact and allows us to neatly represent non-standard concepts like symbolic values and containment.

[3] For a formal definition of heap satisfiablity see our Technical Report (Al-Sibahi et al. 2016)

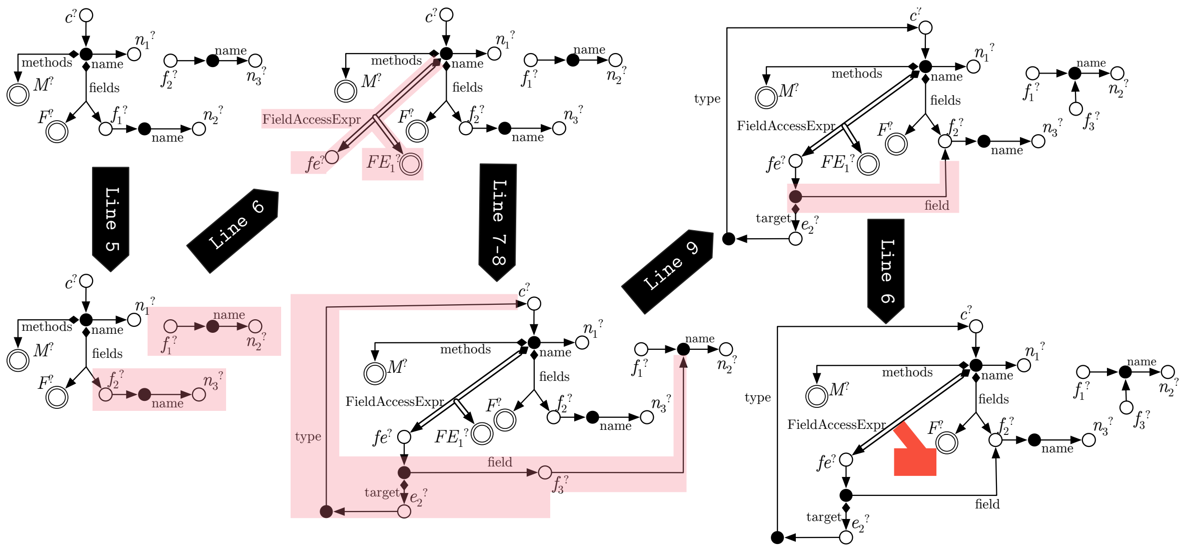


Figure 6: One of the paths when symbolically executing the example Rename-Field refactoring, starting with the symbolic state presented in Fig. 5. Double-stroked arrows represent deep containment constraints.

- Then we perform a deep matching on line 6, prompting the symbolic executor to create a deep containment constraint, represented by a double-stroked arrow ($\Rightarrow$), with type FieldAccessExpr assigning a symbolic set reference $FE_0^?$ (not shown in Figure) to the location assigned to $c^?$.
- In order to iterate over the elements of $FE_0^?$, we non-deterministically chose to partition it into disjoint symbol $fe^?$ and symbolic set $FE_1^?$, executing the body of the foreach with *faexpr* assigned to $fe^?$.
- To check the condition of the if-statement at lines 7-8, we perform a couple of field accesses, which triggers lazy initialization to creates two new symbolic instances: one which is assigned to the symbolic reference $fe^?$ and one which is assigned to its *target* field.
- We non-determinstically chose to execute the then branch—further constraining the values of the fields of $fe^?$—executing the field update statement at line 9, which updates the field access expression to point at the new renamed field instances.
- Finally, we are ready for another iteration at line 6, and this time non-deterministically chose to stop, further constraining $FE_1^?$ to be $\emptyset$ (thus disappearing in the figure).

We shall now define how these (and other execution steps) are realized. We start discussing the *basics* of the presentation format and the simple rules. Then we proceed to the four major ideas in our symbolic executor: *lazy initialization* during heap access and modification, *containment handling* when updating containment links, *lazy iteration* in foreach-loops, and *deep containment constraints* for handling matching expressions.

Basics. During the execution we maintain a store σ mapping variable names to symbolic set expressions. We use two symbolic evaluation functions for TRON's set ($\mathcal{E}[\![\bar{e}]\!]\sigma = e$) and Boolean expressions ($\mathcal{B}[\![\bar{b}]\!]\sigma = b$). They take concrete expressions with a store, and return resulting symbolic expressions by syntactically substituting all variables with their symbolic values as defined by σ. For example, we have $\mathcal{B}[\![x \subseteq y]\!][x \mapsto \{x^?\} \cup \{z^?\}, y \mapsto Y^?] = \{x^?\} \cup \{z^?\} \subseteq Y^?$.

The main judgement has the following format: $\langle s, \sigma, h\rangle \longrightarrow \langle \sigma', h'\rangle$, denoting that the statement s evaluated in the symbolic store σ and heap h, produces a new symbolic store σ' and heap h'.The symbolic execution rules for most statements—assignments, sequencing, branching, object creation and the fix-loop—are standard[4]; for example, consider the true case for if-statements:

$$\text{IfT}\ \frac{\langle s_1, \sigma, h \wedge \mathcal{B}[\![\bar{b}]\!]\sigma\rangle \longrightarrow \langle \sigma', h'\rangle}{\langle \mathsf{if}\ \bar{b}\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2, \sigma, h\rangle \longrightarrow \langle \sigma', h'\rangle}$$

The rule states that one way of symbolically executing an if-statement is by executing its true branch statement s_1 under the assumption that the condition b holds (so $\mathcal{B}[\![\bar{b}]\!]\sigma$ is added to the constraints of the symbolic heap h). Note how the branching rules are non-deterministic; the non-determinism corresponds to branching (back-tracking) in the symbolic executor. Satisfiability of h is an implicit premise in all the rules, and so execution continues only as long as the heap is satisfiable.

Loops with unbounded iteration count give rise to infinite paths in symbolic execution (for instance by considering larger and larger inputs). In order for the symbolic execution algorithm to terminate, we bound the number of paths to be explored.

[4] The formal definitions of our rules and auxiliary functions is in our Technical Report (Al-Sibahi et al. 2016)

Lazy Initialization and Field Access. The rule for symbolic execution of field access is as follows:

$$\textsc{Acc}\;\frac{\begin{array}{c}\mathbf{singleton}\,(\mathcal{E}[\![\overline{e}]\!]\sigma, h) \ni \langle x^?, h''\rangle \\ \mathbf{inst}(x^?, h'') \ni \langle o, h'\rangle \qquad h' = \langle z', \ell', d', \Gamma', b'\rangle\end{array}}{\langle x := \overline{e}.f, \sigma, h\rangle \longrightarrow \langle \sigma[x \mapsto \ell'(o, f)], h'\rangle}$$

Symbolically executing a field access $x := \overline{e}.f$ requires three steps. The first step is to symbolically evaluate $\overline{e}$ to a symbolic set expression e, and then using the **singleton** function to get a single symbol $x^?$ representing the value of e; if e is not already a single symbol, then the **singleton** function will generate a fresh symbol $x^?$ with the correct type and add the constraint $e = x^?$ to the heap, returning a new heap if satisfiable. The second step is to lazily assign a symbolic instance o to $x^?$ (if not already assigned) using the **inst** function, which non-deterministically either creates a new symbolic instance o with the right type and shape, or picks an existing symbolic instance o with compatible type bounds to treat aliasing. The last step is to look up the value of f of the assigned symbolic instance o in the spatial part of the heap ℓ' assigning the resulting value to variable x.

Containment and Field Updates. The symbolic execution of a field update statement $\overline{e_1}.f := \overline{e_2}$ follows a similar pattern to field access:

$$\textsc{Upd}\;\frac{\begin{array}{c}\mathbf{singleton}\,(\mathcal{E}[\![\overline{e}_1]\!]\sigma, h) \ni \langle x^?, h'''\rangle \\ \mathbf{inst}(x^?, h''') \ni \langle o, h''\rangle \\ \mathbf{update}(o, f, \mathcal{E}[\![\overline{e}_2]\!]\sigma, h'') = h'\end{array}}{\langle \overline{e}_1.f := \overline{e}_2, \sigma, h\rangle \longrightarrow \langle \sigma, h'\rangle}$$

After evaluating and resolving $\overline{e}$ to a symbolic instance o, the **update** function is used to update field f of o to point to the evaluated value of $\overline{e}_2$ in the spatial constraints:

$$\mathbf{update}(o, f, e, \langle z, \ell, d, \Gamma, b\rangle) = \begin{cases}\langle z, \ell[\langle o, f\rangle \mapsto e], d, \Gamma, b\rangle & \\ \qquad\qquad\qquad \textbf{if } f \in \mathrm{Field}_{\rightsquigarrow} & \\ \langle z, \ell'[\langle o, f\rangle \mapsto e], d', \Gamma', b \wedge b'\rangle & \\ \qquad\qquad\qquad \textbf{if } f \in \mathrm{Field}_{\blacklozenge} & \end{cases}$$
$$\begin{aligned}\textbf{where } \ell' &= \mathbf{disown}(e, \ell) \\ \langle d', \Gamma', b'\rangle &= \mathbf{dc\text{-}containment}(e, c, z, d, \Gamma)\end{aligned}$$

If f is a containment field we must further ensure that o is the unique owner of e which **update** does by calling **disown** and **dc-containment**.

$$\mathbf{disown}(e, \ell) = \big[\langle o, f\rangle \mapsto \mathbf{do\text{-}f}(e', e) \big| \langle\langle o, f\rangle, e'\rangle \in \mathrm{graph}\,\ell\big]$$
$$\textbf{where } \mathbf{do\text{-}f}(e', e) = \begin{cases} e' & \textbf{if } f \in \mathrm{Fields}_{\rightsquigarrow} \\ e' \setminus e & \textbf{if } f \in \mathrm{Fields}_{\blacklozenge}\end{cases}$$

The **disown** function presented above modifies each containment link in the spatial constraints ℓ to exclude the target symbolic expression e. The **dc-containment** function analogously first excludes e from all deep containment constraints to ensure that there are no stale references to the values of e, and then tries to correctly propagate the effects of the assignment of e back to the containment constraints. The latter is done as follows: for every containment constraint $d(o, c) = e'$ with the same type as e or a super-type of it, we generate a new set symbol $X^?$ with the target type, replace e' with it and then add the constraint $X^? = e' \vee X^? = e \cup e'$ to the heap, which signifies that e might have been added to the deep containment constraints of o; this highlights an interesting interaction between containment links and deep containment constraints which is not immediately obvious, but is necessary to maintain consistency while still keeping a high-level of symbolic abstraction. For subtypes of c, we do almost the same but use the constraint $(X^? = e' \vee X^? = e' \cup Y^?) \wedge e = Y^? \uplus Z^?$ instead, where $Y^?$ and $Z^?$ are fresh set symbols with $Y^?$ having type c and $Z^?$ having the type of e excluding c (in a type bound); this ensures that we refer to all elements in e of type c and only those.

Lazy Iteration with First-class Set Expressions. Two novel ideas of ours are first-class symbolic set expressions, and *lazy iteration* over these. In particular, consider the operational rule for the foreach-loop below:

$$\textsc{For}\;\frac{\begin{array}{c}\mathbf{init}(\overline{me}, h) = \langle \overline{e}, \varsigma\rangle \\ x \mapsfrom \mathcal{E}[\![\overline{e}]\!]\sigma \vdash \langle s, \sigma, h, \varsigma\rangle \xrightarrow{\text{each}} \langle \sigma', h', \varsigma'\rangle\end{array}}{\langle \mathsf{foreach}\ x \in \overline{me}\ \mathsf{do}\ s, \sigma, h\rangle \longrightarrow \langle \sigma', h'\rangle}$$

The first step is to use the **init** function to get the expression $\overline{e}$ to be iterated over, and initialize a control state ς used during iteration depending on the kind of matching expression $\overline{me}$ provided; we will treat ς abstractly for now, and define it precisely later in this section. The other step is to use the $\xrightarrow{\text{each}}$-judgement to iterate over the values of e depending on ς, executing the foreach-body s at each iteration. The two rules for the $\xrightarrow{\text{each}}$-judgement, are provided below:

$$\textsc{ForB}\;\frac{\mathbf{next}(e, h, \varsigma) \ni \langle \mathsf{break}, h'\rangle}{x \mapsfrom e \vdash \langle s, \sigma, h, \varsigma\rangle \xrightarrow{\text{each}} \langle \sigma, h', \varsigma\rangle}$$

$$\textsc{ForC}\;\frac{\begin{array}{c}\mathbf{next}(e, h, \varsigma) \ni \langle \mathsf{cont}\langle x^?, e', \varsigma''\rangle, h'''\rangle \\ \langle s, \sigma[x \mapsto x^?], h'''\rangle \longrightarrow \langle \sigma'', h''\rangle \\ x \mapsfrom e' \vdash \langle s, \sigma'', h'', \varsigma''\rangle \xrightarrow{\text{each}} \langle \sigma', h', \varsigma'\rangle\end{array}}{x \mapsfrom e \vdash \langle s, \sigma, h, \varsigma\rangle \xrightarrow{\text{each}} \langle \sigma', h', \varsigma'\rangle}$$

Both of the above rules depend on the **next** function which target expression e, current control state ς and heap h provides a set of possible next actions; a possible action is either of form $\langle \mathsf{break}, h'\rangle$ which signals that iteration should stop in heap h', or is of form $\langle \mathsf{cont}\langle x^?, e', \varsigma'\rangle, h'\rangle$ which signals that an iteration should happen with symbol $x^?$, afterwards continuing iteration over e' (disjoint from $x^?$) in the new control state ς' and heap h'. The first rule of the $\xrightarrow{\text{each}}$-judgement check whether the set of next possible actions include break and if so it will stop iteration with possibly updated heap h'. The second rule checks whether cont is a possible next action, then executes the loop body s with $x^?$ bound to the range variable x, finally continuing iteration over e' in the updated states.

Now, observe how laziness is achieved with two key ideas: we never explicitly concretize $\overline{me}$, leaving the level of concretization required to be decided by the $\mathbf{next}$ function according to the control state ς, and we iterate using a symbolic reference $x^?$ without requiring an assignment of a symbolic instance (to treat possible aliasing) as this point. Furthermore, by parameterizing the rules for foreach over functions $\mathbf{init}$ and $\mathbf{next}$, it would be easy to add new kinds of expressions without affecting the rules.

Type-Directed Matching with Containment Constraints. We will now discuss how the control state ς and functions $\mathbf{init}$ and $\mathbf{next}$ interact with matching expressions. We define the control state as follows:

$$\varsigma^? ::= \text{ns} \mid \text{ms}\langle c\rangle \mid \text{ms}^*\langle c, e, d\rangle$$

Each alternative stores the required state to execute a given matching expression: ns is used for ordinary iteration, $\text{ms}\langle c\rangle$ is used for shallow matching of elements against c and $\text{ms}^*\langle c, e, d\rangle$ is used for deep matching of elements against type c, storing possibly more elements to be iterated over in e and a copy of deep containment constraints d; the copy of containment constraints is kept in order to retrieve the deep containment constraint values that were available before iteration, which would represent the concrete objects that would have been matched by a concrete deep match operation. The $\mathbf{init}$ is therefore defined to map each expression to its initial control state:

$$\mathbf{init}(\overline{e}, h) = \langle \overline{e}, \text{ns}\rangle \quad \mathbf{init}(\overline{e}\ \mathsf{match}\ c, h) = \langle \overline{e}, \text{ms}(c)\rangle$$
$$\mathbf{init}(\overline{e}\ \mathsf{match}^*\ c, \langle z, \ell, d, \Gamma, b\rangle) = \langle \overline{e}, \text{ms}^*(c, \emptyset, d)\rangle$$

The $\mathbf{next}$ function is more interesting since it calculates the possible next actions for iteration. For straightforward iteration, the next function is defined as follows:

$$\mathbf{next}(e, h, \text{ns}) = \{\langle \text{break}, (h \wedge e = \emptyset)\rangle | (h \wedge e = \emptyset) \text{ sat}\} \cup$$
$$\left\{\langle \text{cont}\langle x^?, X^?, \text{ns}\rangle, h'\rangle \middle| \mathbf{partition}(e, h) = \langle x^?, X^?, h'\rangle\right\}$$

It states that there are two possible actions: we can stop iterating if it is possible to constraint e to be $\emptyset$, and we can try to use $\mathbf{partition}$ to split e into a symbol $x^?$ and a disjoint set symbol $X^?$ and then continue iteration with that. The $\mathbf{partition}$ function essentially generates fresh symbol $x^?$ and set symbol $X^?$ with the right types adding the constraint $e = x^? \uplus X^?$ to h, returning a new heap if valid.

For matching iteration, $\mathbf{next}$ is defined as follows:

$$\mathbf{next}(e, h, \text{ms}\langle c\rangle) = \{\langle \text{break}, (h \wedge e = \emptyset)\rangle | (h \wedge e = \emptyset) \text{ sat}\} \cup$$
$$\left\{\langle \text{break}, h'\rangle \middle| \begin{array}{r}\mathbf{partition}(e, h) = \langle x^?, X^?, h''\rangle \wedge \\ \mathbf{match}(x^?, X^?, c, h'') \ni \langle \text{ff}, h'\rangle\end{array}\right\} \cup$$
$$\left\{\begin{array}{r}\langle \text{cont}\langle x^?, X^?, \\ \text{ms}\langle c\rangle\rangle, h'\rangle\end{array} \middle| \begin{array}{r}\mathbf{partition}(e, h) = \langle x^?, X^?, h''\rangle \wedge \\ \mathbf{match}(x^?, X^?, c, h'') \ni \langle \text{tt}, h'\rangle\end{array}\right\}$$

In this control state, there are up to three possible actions: one where e is constraint to $\emptyset$, and two where $\mathbf{partition}$ is used to get $x^?$ and $X^?$, which are then matched against $c^?$ using $\mathbf{match}$. The $\mathbf{match}$ function returns a set of states each indicating whether matching $x^?$ against c was successful: if $\langle \text{tt}, h'\rangle$ is included then h' constraints the type of $x^?$ to be a subtype of c, and if $\langle \text{ff}, h'\rangle$ is included then h' constraints the type bounds of $x^?$ and $X^?$ to exclude c as a possible supertype. A match is always successful if the type of $x^?$ is a subtype of c, always fails when the c is unrelated to or excluded from type bounds of $x^?$, and allows both when the type of $x^?$ is a supertype of c.

Finally, the definition of $\mathbf{next}$ for deep matching is:

$$\mathbf{next}(e_0, h_0, \text{ms}^*\langle c, e_0', d\rangle) = (\text{lfp}\ \Phi \mapsto \mathbf{next\text{-}ms}_\Phi)(e_0, e_0', h)$$
$$\text{where } \mathbf{next\text{-}ms}_\Phi(e, e', h') = \mathbf{next}(e', h \wedge e = \emptyset, \text{ns}) \cup$$
$$\bigcup \left\{\Phi(X^?, e' \cup e'', h') \middle| \begin{array}{r}\mathbf{partition}(e, h) = \langle x^?, X^?, h'''\rangle \wedge \\ \mathbf{match}(x^?, X^?, c, h''') \ni \langle \text{ff}, h''\rangle \wedge \\ \mathbf{dcs}(x^?, c, d, h'') \ni \langle e'', h'\rangle\end{array}\right\} \cup$$
$$\left\{\begin{array}{r}\langle \text{cont}\langle x^?, X^?, \\ \text{ms}^*\langle c, e' \cup e'', d\rangle\rangle, h'\rangle\end{array} \middle| \begin{array}{r}\mathbf{partition}(e, h) = \langle x^?, X^?, h'''\rangle \wedge \\ \mathbf{match}(x^?, X^?, c, h''') \ni \langle \text{tt}, h''\rangle \wedge \\ \mathbf{dcs}(x^?, c, d, h'') \ni \langle e'', h'\rangle\end{array}\right\}$$

There are again three possible actions in this control state. The first is to try to constraint e to $\emptyset$ like in the other cases, but this time we must continue to iterate over e' which was used to collect deep containment constraint values during iteration. The second possible action is to use $\mathbf{partition}$ and $\mathbf{match}$ on c, where the match was unsuccesful; we use the $\mathbf{dcs}$ function to assign a location o to $x^?$ and lookup the deep containment constraint value $d(o, c) = e''$ (creating it in the provided heap if non-existing), then adding it to the control state and continue iterating over the rest $X^?$; note that since $x^?$ did not match the target type c we do not consider it for iteration. The final possible action is where the match was successful and so we use the $x^?$ for the next iteration; we still need to consider possible descendants of $x^?$ of type c and so use $\mathbf{dcs}$ to get the deep containment constraint value and add it to the control state. Observe how the use of deep containment constraints allows us to provide a higher-level abstraction over structures focusing only on instances of the target type, and without explicitly considering all intermediate shapes of data.

5. Evaluation

We implemented our technique in a prototype tool[5], which we evaluate to show the concrete benefits of our technique.

5.1 Test Generation

White-box test generation is a classical application of symbolic execution, and we aim to use it as an example for evaluating our symbolic execution algorithm. We have built a white-box test generator and compare its effectiveness against a baseline black-box test generator.

We aim to compare the test generators according to their effectiveness, which is how well a test suite exercises the

[5] `https://github.com/models-team/SymexTRON`

transformation-under-test (TUT). We use branch coverage as the target metric, which we define for TRON constructs as follows: for `if`-statements both branches must be taken, for `fix`-loops we check whether it is run one or more times[6] and for `foreach`-loops we check whether it is run zero, one or more times.

White-box Test Generator. The white-box test generator is a simple extension of the symbolic execution algorithm presented in Sect. 4, requiring only two new additions:

1. Memoising a copy of the spatial constraints which values are not modified by field updates, thus keeping track of the initial structure of input.
2. A translator between the output model given by the model finder to concrete data usable by the target TRON program.

Black-box Test Generator. The black-box test generator optimizes towards *meta-model coverage*, which is the literature-recommended metric (Wang et al. 2006; Finot et al. 2013). A test suite is said to have full meta-model coverage if each subtype of relevant classes is present in at least in one test case, and each relevant field is instantiated with each valid multiplicity (i.e. zero, one or many).

Subject Programs. The subject programs were selected according to three criteria: be an interesting representative variety of realistic transformations, be independently specified to avoid bias, and be feasible to implement in TRON. To fulfill the first criterion, we chose transformations from two categories: model transformations and refactorings. For the second criterion, we ported the model transformations from the ATL transformation zoo[7] and chose the refactorings from Fowler's classic collection (Fowler 1999). The third criterion is achieved by picking suitably sized transformations that satisfy our resource and design constraints, since it takes time to manually port complex transformations correctly (despite language similarity) and TRON lacks abstractions for modularity that full languages have. We ended up with 3 model transformations and 4 refactorings, all of which we describe below.

Refactorings.

- An extended version of the *Rename field* refactoring used as our running example.
- *Rename method*: renames a target method in a class, and ensures that all calls to the correct overloading of this method (with the right types) must refer to the updated name.
- *Extract superclass*: creates a common superclass of two classes with similar structure ensuring that all common fields are pulled up and that both classes inherit from it.
- *Replace delegation with inheritance*: Makes a class inherit directly from a type instead of using a field for delegation, updating all method calls targeting that field to use `this` as a target instead.

Model Transformations.

- The *Families to Persons* model transformation (Fam2Pers), converts a model of a traditional family with a mother, father and possibly children to a collection of individuals with explicit gender (male or female).
- The classical *Class to Relational* model transformation (Class2Rel), which converts an object-oriented class model to a relational database schema.
- The *Path expression to Petri net* model transformation (Path2Petri), which converts a path expression with states and transitions to a full Petri net with named places, different types of arcs and weighted transitions.

The source code of all the programs is included in our associated Technical Report (Al-Sibahi et al. 2016).

Porting Transformations to TRON. By design TRON is a minimal language, and so there are non-core transformation languages features that must be handled when porting transformations from fully-featured languages to TRON. In particular, three features had to be handled for the considered subject programs: functions, implicit tracing links and circular data dependencies (the latter two present in model transformation languages like ATL). When a transformation is ported TRON one must take care to correctly inline function calls, which is done by replacing the calls with the function body, substituting the parameters with the provided arguments, renaming local variables to avoid clashes, and converting any explicit recursion to use the '`foreach`'-statement or '`fix`'-statement. To handle tracing links one must take care to augment the meta-model to include them explicitly, and to explicate assignment to the tracing links on object creation in the transformation; for circular data dependencies one must ensure to separate the object creation phase from the actual translation phase.

Set-up. The experiment was set-up to automatically run both test generators automatically on all the described subject programs. The white-box test generator was bounded in the number of iterations (2, except for Fam2Pers which uses 3 due to meta-model constraints) and instances considered by the model finder (6 for model transformations, 10 for refactorings), and a reasonable time-out of 1 hour was put in place. We also added light-weight support for bidirectional fields in the model finder to better support the model transformations. The prototype symbolic executor, was implemented in Scala 2.11.7 (Odersky and Rompf 2014) and the evaluation was run on a 2.3 GHz Core i7 MacBook Pro (OS X 10.11). The external model finder KodKod was configured to use the parallel SAT solver Plingeling (Biere 2014).

Test Generation Results. We ran a series of toy programs exercising the various constructs of TRON as a warm up for

[6] `fix`-loops must run the body at least once, according to the semantics

[7] `https://www.eclipse.org/atl/atlTransformations/`

our test generators; the white-box test generator achieved 100% code coverage for all programs beating the black-box test generator, and all under 30 seconds of execution time.

Table 2 shows the results of running the test generators on the selected subject programs (model transformations and refactorings).

Refactorings. The white-box test generator achieves better code coverage than the baseline black-box test generator for all the refactorings, reaching 100% coverage for two. We hypothesise that refactorings do many targeted modifications of complex models, making it hard to generate tests that cover the required parts without access to the transformation code; The test generated by black-box had high meta-model coverage (see Tbl. 2) but did not fully exercise the transformation, in contrast to the more focused tests generated by the white-box test generator.

Due to the nature of symbolic execution, the white-box test generator was unsurprisingly slower than the black-box test generator. We believe that the higher precision in bug finding offsets the runtime cost; test generation is an occasional offline task and 1 computer hour is not unreasonable to use compared to the many hours a programmer would otherwise have spent on the same task.

Model Transformations. The white-box test generator achieved good results for model transformations, performing better than the black-box test generator for the Fam2Pers and Class2Rel transformations, and worse for the Path2Petri transformation. We suspect that the black-box test generator performs well on model transformations because they primarily translate structured data according to their meta-model types, without relying on complex constraints and cases. Therefore, having a test suite with high meta-model coverage will generate the necessary different types to trigger the right execution paths resulting in acceptable branch coverage. The white-box test generator performed less impressively on model transformations than refactorings because the symbolic executor did not reach the right paths before the time-out triggered. The sequential composition of complex for-loops results in an explosion of paths to be explored, and so it takes significantly more time to explore the whole program and generate interesting tests.

Comparing Coverage Criteria. The black-box test generator optimizes towards achieving maximal meta-model coverage (see Tbl. 2), but seems to achieve mixed branch coverage results, and although it performed better for model transformations than for refactorings, it never reached full branch coverage. This indicates that there is little guarantee that high meta-model coverage ensures high branch coverage, which is an interesting experience for building both white-box and black-box tools.

The symbolic executor achieved high branch coverage for most subjects without achieving the same high meta-model coverage—the lowest being 31.03% meta-model coverage for the ExtractSuper refactoring that we achieved full branch coverage for—which indicates that there is no correlation the other way as well. It would of course require a more extensive empirical study to conclusively affirm our hypothesis.

5.2 Comparison with Symbolic Executors for Object-Oriented Languages

Two of the best known and well-supported symbolic executors for object-oriented programming languages are Symbolic PathFinder (Pasareanu et al. 2013) and Microsoft IntelliTest/Pex (Tillmann and de Halleux 2008). We will describe our experiences trying to encode various high-level transformation features—sets, containment and deep matching—and the difficulties faced when these features are not handled first-class.

Symbolic Support for Sets. The first challenge we faced was how to symbolically encode sets in traditional symbolic executors. Symbolic PathFinder does not directly support standard Java collections like HashSet or TreeSet, and using those collections during symbolic execution leads to errors: a symbol does not have a hash value, and it is not possible to symbolically compare two objects. One could try a more cumbersome encoding by using lists and handling inequality constraints explicitly, but it is unclear how to generate interesting instances of such sets automatically. Pex tries to dynamically construct instances of sets using arrays, but it is hard in practice to make it generate an array of distinct elements that is usable to construct an interesting set.

We treat set values first-class which exploits the support of set theories in model finders like KodKod and SMT solvers (Kröning et al. 2009). We believe that implementing this could be beneficial for these traditional symbolic executors as well.

Enforcing Deep Containment Constraints. Another challenge is enforcing containment-like constraints with acyclicity and non-sharing for abstract syntax trees. Traditional symbolic executors support deep containment constraints neither directly nor indirectly. Hypothetically, acyclicity constraints could be enforced statically by giving all objects unique identifiers and fixing an ordering between contained objects and parents; however, this requires both the management of a complex system on top of existing dynamic structures, and it is unclear how to efficiently handle dynamic to such links. In contrast, first class support of these constraints in TRON makes it easy to handle dynamic updates and allows specialized techniques to be used to handle such constraints.

Deep Matching and Visitors. Deep matching is straightforward to encode using reflection, but that approach is not handled well by symbolic executors, and so is to be avoided. Traditionally traversal of abstract syntax is done using recursive visitors, which uses plain classes and thus is better supported.

However, this approach is non-optimal from a symbolic execution point of view, since to reach the relevant part of an

Program	LOC	Meta-model coverage (%)		Branch coverage (%)		Time (s)	
		Black-box	White-box	Black-box	White-box	Black-box	White-box
RenameField	53 + 12 = 65	96.55	70.69	60.00	100.00	141.5	336.6
RenameMethod	53 + 28 = 81	96.55	93.10	26.67	93.33	141.7	3600.0
ExtractSuper	53 + 29 = 82	98.28	31.03	75.00	100.00	124.8	386.4
ReplaceDelegation	53 + 30 = 83	100.00	85.19	47.06	76.47	115.5	3600.0
Fam2Pers	21 + 56 = 77	100.00	88.00	100.00	100.00	4.8	135.4
Path2Petri	42 + 58 = 100	100.00	37.50	88.89	33.33	1.8	3600.0
Class2Rel	34 + 100 = 134	100.00	100.00	70.83	75.00	3.8	3600.0

Table 2: Results of running the test generators on subject programs. Here LOC indicates lines of code, where the first component of the summation is the size of the data model and the second component is the size of the transformation.

abstract syntax tree—like a field access expression—one has to consider all intermediate shapes—i.e., classes, methods, different kinds of statements and containing expressions in our example—which hits a combinatorial explosion, even with reasonably small bounds.

In contrast, we abstract away intermediate shapes with deep containment constraints, which allows reasoning about only the parts of the data structure we are interested in. Transformations like refactorings often perform local changes on the abstract syntax trees, and so this approach seems especially beneficial in those cases.

5.3 Threats and Limitations

The main threat to validity of the experiment is that we implemented the subject programs ourselves, introducing the possibility of bias and errors in the implementation. For the model transformations, we mitigated this by choosing existing ones from ATL, and for the refactorings we chose a number of standard ones from Fowler's authoritative book (Fowler 1999). Furthemore, minor implementation mistakes are of lesser importance since the number of found errors is not an evaluation criterion. Inozemtseva and Holmes (Inozemtseva and Holmes 2014) show that test coverage is not a strongly correlative measure for effectiveness. However, arguably a test suite which has low code coverage is going to miss bugs because it simply does not visit code present in some of the branches. The black-box test generator has been implemented by us optimizing for the standard meta-model coverage metric (Finot et al. 2013; Wang et al. 2006) to avoid bias, since we could not find an existing third-party tool that was suitable for our purposes. We are not experts on Symbolic PathFinder and Pex, and could have missed better ways to encode high-level features. We mitigated this by systematically reading the available documentation, and searching on forums and mailing lists for answers to similar challenges.

6. Related Work

Symbolic Execution of High-level Transformation Languages. Simple symbolic execution algorithms (Lucio and Vangheluwe 2013) exist for significantly less expressive transformation languages like DSLTrans, which bounds loops and does not permit dependent state and loop iterations. This lack of expressiveness allows the symbolic executor to be heavily specialized and quick, but the algorithm is hard to generalize for more expressive Turing-complete languages like TRON.

More complex whitebox-based algorithms are presented by ATLTest (González and Cabot 2012) and TETRA Box (Schönböck et al. 2013). These tools only support a class of transformations that can not modify input state. Therefore, it is not possible to easily express complex transformations like the refactorings considered in this paper. Furthermore, the method presented in this paper, is fully-formalized and evaluated, showing applicability of our framework for a broader range of transformations.

Test Generation for Transformations. The latest survey on verification of model transformations (Rahim and Whittle 2015) shows that most test generation techniques for model transformations focus on black-box testing, which do not account for concrete transformation semantics and thus may fail to cover program statements as shown in our evaluation. There is a test generation tool for Maude (Riesco 2010, 2012) based on bounded narrowing (practically, symbolic execution for rewriting languages). However, complex transformations are hard to write in the style of term-rewriting systems—since they modify object graphs—and the object-oriented extension is not as far as we understand supported by the test generator. A black-box test generation tool called Dolly (Mongiovi et al. 2014) is used to test C and Java based refactoring engines with promising results. As our evaluation results indicate that white-box based techniques have better effectiveness than black-box based ones, it could be interesting to see whether we could adapt some of our novel ideas for a language like Java and increase the number of bugs found.

7. Conclusion

We have presented a symbolic execution technique for high-level transformation languages, which we have formalized using our representative language TRON. Our evaluation shows that not only is symbolic execution feasible for expressive high-level transformations, but that it also achieves good coverage when used for test generation.

References

A. S. Al-Sibahi, A. S. Dimovski, and A. Wąsowski. Symextron: Symbolic execution of high-level transformations. Technical Report TR-2016-196, IT University of Copenhagen, Denmark, 2016.

A. Biere. Yet another local search solver and lingeling and friends entering the sat competition 2014. In *SAT Competition 2014, Vienna, Austria, July 14-17, Proceedings*, page 2, 2014.

M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/xt 0.17. A language and toolset for program transformation. *Sci. Comput. Program.*, 72(1-2):52–70, 2008. doi: 10.1016/j.scico.2007.11.003. URL `http://dx.doi.org/10.1016/j.scico.2007.11.003`.

C. Cadar and A. F. Donaldson. Analysing the program analyser. In L. K. Dillon, W. Visser, and L. Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, pages 765–768. ACM, 2016. ISBN 978-1-4503-4205-6. doi: 10.1145/2889160.2889206. URL `http://doi.acm.org/10.1145/2889160.2889206`.

J. R. Cordy. The TXL source transformation language. *Sci. Comput. Program.*, 61(3):190–210, 2006. doi: 10.1016/j.scico.2006.04.002. URL `http://dx.doi.org/10.1016/j.scico.2006.04.002`.

X. Deng, J. Lee, and Robby. Efficient and formal generalized symbolic execution. *Autom. Softw. Eng.*, 19(3):233–301, 2012. doi: 10.1007/s10515-011-0089-9. URL `http://dx.doi.org/10.1007/s10515-011-0089-9`.

O. Finot, J. Mottu, G. Sunyé, and T. Degueule. Using metamodel coverage to qualify test oracles. In B. Baudry, J. Dingel, L. Lucio, and H. Vangheluwe, editors, *Proceedings of the Second Workshop on the Analysis of Model Transformations (AMT 2013), Miami, FL, USA, September 29, 2013*, volume 1077 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2013. URL `http://ceur-ws.org/Vol-1077/amt13_submission_3.pdf`.

M. Fowler. *Refactoring - Improving the Design of Existing Code*. Addison Wesley object technology series. Addison-Wesley, 1999. ISBN 978-0-201-48567-7. URL `http://martinfowler.com/books/refactoring.html`.

C. A. González and J. Cabot. Atltest: A white-box test generation approach for ATL transformations. In R. B. France, J. Kazmeier, R. Breu, and C. Atkinson, editors, *Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings*, volume 7590 of *Lecture Notes in Computer Science*, pages 449–464. Springer, 2012. ISBN 978-3-642-33665-2. doi: 10.1007/978-3-642-33666-9_29. URL `http://dx.doi.org/10.1007/978-3-642-33666-9_29`.

C. A. R. Hoare. The verifying compiler, a grand challenge for computing research. In R. Cousot, editor, *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings*, volume 3385 of *Lecture Notes in Computer Science*, pages 78–78. Springer, 2005. ISBN 3-540-24297-X. doi: 10.1007/978-3-540-30579-8_5. URL `http://dx.doi.org/10.1007/978-3-540-30579-8_5`.

L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In P. Jalote, L. C. Briand, and A. van der Hoek, editors, *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 435–445. ACM, 2014. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568271. URL `http://doi.acm.org/10.1145/2568225.2568271`.

F. Jouault and I. Kurtev. Transforming models with ATL. In J. Bruel, editor, *Satellite Events at the MoDELS 2005 Conference, MoDELS 2005 International Workshops, Doctoral Symposium, Educators Symposium, Montego Bay, Jamaica, October 2-7, 2005, Revised Selected Papers*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer, 2005. ISBN 3-540-31780-5. doi: 10.1007/11663430_14. URL `http://dx.doi.org/10.1007/11663430_14`.

L. C. L. Kats and E. Visser. The spoofax language workbench: rules for declarative specification of languages and ides. In W. R. Cook, S. Clarke, and M. C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 444–463. ACM, 2010. ISBN 978-1-4503-0203-6. doi: 10.1145/1869459.1869497. URL `http://doi.acm.org/10.1145/1869459.1869497`.

S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In H. Garavel and J. Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, volume 2619 of *Lecture Notes in Computer Science*, pages 553–568. Springer, 2003. ISBN 3-540-00898-5. doi: 10.1007/3-540-36577-X_40. URL `http://dx.doi.org/10.1007/3-540-36577-X_40`.

J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976. doi: 10.1145/360248.360252. URL `http://doi.acm.org/10.1145/360248.360252`.

D. S. Kolovos, R. F. Paige, and F. Polack. The epsilon transformation language. In A. Vallecillo, J. Gray, and A. Pierantonio, editors, *Theory and Practice of Model Transformations, First International Conference, ICMT 2008, Zürich, Switzerland, July 1-2, 2008, Proceedings*, volume 5063 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2008. ISBN 978-3-540-69926-2. doi: 10.1007/978-3-540-69927-9_4. URL `http://dx.doi.org/10.1007/978-3-540-69927-9_4`.

D. Kröning, P. Rümmer, and G. Weissenbacher. A proposal for a theory of finite sets, lists, and maps for the smt-lib standard. In *Informal proceedings, 7th International Workshop on Satisfiability Modulo Theories at CADE. Vol. 22. 2009*, 2009.

L. Lucio and H. Vangheluwe. Symbolic Execution for the Verification of Model Transformations. *VOLT@STAF*, pages 1–29, Apr. 2013.

N. Mitchell and C. Runciman. Uniform boilerplate and list processing. In G. Keller, editor, *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2007, Freiburg, Germany, September 30, 2007*, pages 49–60. ACM, 2007. ISBN 978-1-59593-674-5. doi: 10.1145/1291201.1291208. URL `http://doi.acm.org/10.1145/1291201.1291208`.

M. Mongiovi, G. Mendes, R. Gheyi, G. Soares, and M. Ribeiro. Scaling testing of refactoring engines. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 371–380. IEEE Computer Society, 2014. ISBN 978-0-7695-5303-0. doi: 10.1109/ICSME.2014.59. URL http://dx.doi.org/10.1109/ICSME.2014.59.

Object Management Group. *Meta Object Facility (MOF) 2.0 Query/View/ Transformation Specification*, jan 2011. URL http://www.omg.org/spec/QVT/.

M. Odersky and T. Rompf. Unifying functional and object-oriented programming with scala. *Commun. ACM*, 57(4):76–86, 2014. doi: 10.1145/2591013. URL http://doi.acm.org/10.1145/2591013.

C. S. Pasareanu, W. Visser, D. H. Bushnell, J. Geldenhuys, P. C. Mehlitz, and N. Rungta. Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. *Autom. Softw. Eng.*, 20(3):391–425, 2013. doi: 10.1007/s10515-013-0122-2. URL http://dx.doi.org/10.1007/s10515-013-0122-2.

L. A. Rahim and J. Whittle. A survey of approaches for verifying model transformations. *Software and System Modeling*, 14(2): 1003–1028, 2015. doi: 10.1007/s10270-013-0358-0. URL http://dx.doi.org/10.1007/s10270-013-0358-0.

A. Riesco. Test-case generation for maude functional modules. In T. Mossakowski and H. Kreowski, editors, *WADT 2010, Etelsen, Germany, July 1-4, 2010*, volume 7137 of *Lecture Notes in Computer Science*, pages 287–301. Springer, 2010. ISBN 978-3-642-28411-3. doi: 10.1007/978-3-642-28412-0_18. URL http://dx.doi.org/10.1007/978-3-642-28412-0_18.

A. Riesco. Using narrowing to test maude specifications. In *WRLA'12, Tallinn, Estonia, March 24-25, 2012*, pages 201–220, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-34004-8. doi: 10.1007/978-3-642-34005-5_11. URL http://dx.doi.org/10.1007/978-3-642-34005-5_11.

M. Schäfer, T. Ekman, and O. de Moor. Challenge proposal: verification of refactorings. In T. Altenkirch and T. D. Millstein, editors, *Proceedings of the 3rd ACM Workshop Programming Languages meets Program Verification, PLPV 2009, Savannah, GA, USA, January 20, 2009*, pages 67–72. ACM, 2009. ISBN 978-1-60558-330-3. doi: 10.1145/1481848.1481859. URL http://doi.acm.org/10.1145/1481848.1481859.

J. Schönböck, G. Kappel, M. Wimmer, A. Kusel, W. Retschitzegger, and W. Schwinger. Tetrabox - A generic white-box testing framework for model transformations. In P. Muenchaisri and G. Rothermel, editors, *20th Asia-Pacific Software Engineering Conference, APSEC 2013, Ratchathewi, Bangkok, Thailand, December 2-5, 2013 - Volume 1*, pages 75–82. IEEE Computer Society, 2013. doi: 10.1109/APSEC.2013.21. URL http://dx.doi.org/10.1109/APSEC.2013.21.

A. Sloane. Lightweight language processing in kiama. In J. Fernandes, R. Lämmel, J. Visser, and J. Saraiva, editors, *Generative and Transformational Techniques in Software Engineering III*, volume 6491 of *Lecture Notes in Computer Science*, pages 408–425. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-18022-4. doi: 10.1007/978-3-642-18023-1_12. URL http://dx.doi.org/10.1007/978-3-642-18023-1_12.

N. Tillmann and J. de Halleux. Pex-white box test generation for .net. In B. Beckert and R. Hähnle, editors, *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer, 2008. ISBN 978-3-540-79123-2. doi: 10.1007/978-3-540-79124-9_10. URL http://dx.doi.org/10.1007/978-3-540-79124-9_10.

E. Torlak and D. Jackson. Kodkod: A relational model finder. In O. Grumberg and M. Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4424 of *Lecture Notes in Computer Science*, pages 632–647. Springer, 2007. ISBN 978-3-540-71208-4. doi: 10.1007/978-3-540-71209-1_49. URL http://dx.doi.org/10.1007/978-3-540-71209-1_49.

J. Troya and A. Vallecillo. A rewriting logic semantics for ATL. *Journal of Object Technology*, 10:5: 1–29, 2011. doi: 10.5381/jot.2011.10.1.a5. URL http://dx.doi.org/10.5381/jot.2011.10.1.a5.

J. Wang, S. Kim, and D. A. Carrington. Verifying metamodel coverage of model transformations. In *17th Australian Software Engineering Conference (ASWEC 2006), 18-21 April 2006, Sydney, Australia*, pages 270–282. IEEE Computer Society, 2006. ISBN 0-7695-2551-2. doi: 10.1109/ASWEC.2006.55. URL http://dx.doi.org/10.1109/ASWEC.2006.55.