RI12007 February 2011 Computer Science

IBM Research Report

Operational Abstractions of Model Transforms

Vibha Singhal Sinha
IBM Research – India

Pankaj Dhoolia
IBM Research – India

Senthil Mani
IBM Research – India

Saurabh Sinha
IBM Research – India

IBM Research Division
Almaden - Austin - Beijing - Delhi - Haifa - T.J. Watson - Tokyo - Zurich

Abstract

Model transforms convert a model to another model or text. Transforms play a key role in model-driven engineering; therefore, it is essential for a transform user to understand the transform semantics. However, examining the transform code to obtain such understanding can be cumbersome and impractical. To address this, we present an operational abstraction of model transforms. The abstraction captures the essential transformation semantics in terms of the structure of the output and the influence of input-model elements on output fragments. Thus, the abstraction supports the transform user's perspective, which is focused on inputs and outputs, and is unconcerned with implementation details. We present an automated approach that uses a combination of selective path enumeration, forced execution of enumerated paths, and an offline trace-merging analysis to synthesize operational abstractions. We discuss different applications of the abstraction. We implemented our approach for XSLT-based model transforms and conducted empirical studies. Our results indicate that selective path enumeration can be highly effective in reducing the scope of path exploration, while ensuring that the abstraction is complete. Our user study illustrates that the abstraction can improve user efficiency in debugging faulty input models. Our final study shows the feasibility of using abstractions for two metamodel management tasks.

1 Introduction

Model transforms [1] are a class of programs that convert structured input (formal models) to structured output (other models or text). Commonly used modeling formalisms include UML, Ecore, and XML. An archetypal model transform is a code generator that automatically generates code (*e.g.,* Java) from a formal model (*e.g.,* a UML class diagram). Transform implementations vary widely: a transform can be implemented in an imperative manner, a declarative way, or a rule-based form. For instance, a transform can be coded in a general-purpose language, such as Java; alternatively, it can be implemented using templating languages, such as XSLT.

When provided with a transform written by someone else, users often struggle in understanding the transform semantics, such as how different elements in the input model contribute to the output. To use a third-party transform effectively and efficiently, it is essential for the user to understand the transformation semantics, in terms of the expected inputs and generated outputs. Often, users rely on the transform documentation and sample input models to get such understanding. The more inquisitive user may delve into the transform implementation, when available, but that can be a cumbersome and time-consuming task—and the typical user may not be well-versed in the myriad of transform implementation technologies. Thus, tools designed to help the users in understanding the semantics of a transform at an abstract level can be very useful.

Transform Abstraction

Our goal is to develop an automated technique for creating operational abstractions of model transforms. The abstraction should capture the essential transformation semantics, while abstracting away the implementation details. The user's perspective of a transform is focused on the inputs and outputs of the transform—the implementation details are irrelevant. Therefore, in essence, we intend the abstraction to support this user perspective.

Figure 1 presents a sample transform, `PrintUser`, shown in a pseudo-code form. The transform takes as input an XML file containing user information and prints the information in a plain-text format. Figure 2(a) shows a sample input model, which contains information about users and their children; Part (b) of the figure shows the output. The input model contains two `<user>` elements (line 2 and lines 3–6). The transform user is concerned primarily with providing valid inputs and obtaining the expected outputs. Thus, the user would be interested in understanding, for example, how input-model entities influence the generation of output fragments. For instance, the input-model attribute `hasMiddle` determines whether the middle name is printed in the output. This is useful information from the

```
   function main() {
 1.   foreach user r in userData do
 2.      print("--- User ---\n")
 3.      call printName(r)
 4.      totalChildren = 0
 5.      if r.children != null then
 6.         totalChildren = r.children.users.length
 7.         print("--- Children ---\n")
 8.         foreach user c in r.children do
 9.            call printName(c)
10.            if c.gender = "M" then
11.               print("Boy, ", c.age)
12.            else if c.gender = "F" then
13.               print("Girl, ", c.age)
               endif
14.            print("\n")
            endfor
15.         print("\n")
         endif
16.      if totalChildren > 0
17.         print("User has children\n")
         endif
      endfor
   }
   function printName(User user) {
18.   print(user.first + " ")
19.   if user.hasMiddle = "true" then
20.      print(user.middle + " ")
      endif
21.   print(user.last)
   }
```

Figure 1: Sample model transform `PrintUser`.

```
1. <userData genId="1">
2.   <user first="Jane" last="Eyre"
      genId="1.2"/>
3.   <user first="Amit" last="Vyas"
      genId="1.4"
4.    gender="M" middle="R"/>
5.     <children genId="1.4.2"/>
6.   </user>
7. </userData>
```

(a)

```
1. --- User ---
2. Jane Eyre
3.
4. --- User ---
5. Amit Vyas
6. --- Children ---
7.
```

(b)

Figure 2: (a) An input model for `PrintUser`, and (b) the corresponding output.

user's perspective because it helps the user understand why the `middle` attribute for the first `<user>` element in the input model (line 4) does not contribute to the output.

Our notion of operational abstraction captures such information. It summarizes the transform output and how input-model entities influence different fragments of the output. From another perspective, the abstraction is a *projection* of the transform, created via propagation of inputs to outputs and filtering intermediate and irrelevant information. We require the abstraction to be complete and sound. Whereas *completeness* ensures that the abstraction captures all relevant paths from the transform, *soundness* guarantees that the abstraction contains no path that does not occur in the transform.

Our Approach

We present a dynamic-analysis-based technique for constructing operational abstractions of transforms. The overall idea underlying the abstraction construction is to (1) enumerate the transform paths that are relevant for analysis; (2) analyze individual paths to compute information, such as how input-model entities are accessed and propagated to the output; and (3) compose together the path-specific information to synthesize the operational abstraction.

In the *first phase*, the technique enumerates transform paths. Because we require the abstraction to be complete,

all acyclic paths must be explored. However, a naive exhaustive exploration can be prohibitively expensive because of the combinatorial blowup in the number of paths, which would render our approach impractical. In fact, as our results indicate (Section 5.2), the problem of exponential number of paths can be quite severe in practice. The combinatorial explosion occurs in the presence of long chains of conditional statements. Our observation is that all combinations of paths through such chains need not be explored—the paths can be traversed selectively (or a set of paths can be explored as group, *e.g.,* as in the notion of "path families" [19]), while *ensuring* that the synthesized abstraction is complete. We provide a characterization of the cases where selective exploration can be performed without compromising completeness.

In the *second phase*, the technique analyzes the enumerated paths. This analysis could be performed statically or dynamically. We perform a dynamic analysis. Specifically, our technique is based on forced exploration of program paths, which is a general dynamic analysis for exploring new program behaviors without generating new test inputs [17, 24, 27]. Our technique adds probes to the transform to force the execution of specific paths at runtime. Given the specification of a path to be traversed, in terms of the desired outcomes of branches, the probes drive execution along the intended branches from conditional statements. During the execution, additional probes collect data about accesses to input-model entities, evaluation of conditionals, and construction of the output.

Finally, in the *third phase*, the technique analyzes the generated execution traces—one trace for each explored path. It analyzes each trace to remove intermediate computation and computation irrelevant to the transform output, combines the traces, and synthesizes the transform abstraction.

The transform operational abstraction can be useful for many applications. The most obvious application is to assist users in transform comprehension, but there are other useful applications as well. For instance, the abstraction can be useful for assessing the coverage of the input metamodel by a transform [25] or the footprint of a transform over the input metamodel or model [12]. Such information can also be useful for pruning metamodels [21], refining metamodels, identifying critical regions in metamodels, and assisting with automated fault repair. Section 4 discusses these applications.

Overview of Results

We implemented our technique for XSLT-based transforms that generate text output (*e.g.,* Java code and configuration files). Using the implementation and a set of real model transforms, we conducted empirical studies to evaluate different aspects of the technique. In the first study, we evaluated the effectiveness of selective path enumeration in reducing the scope of path exploration. Our approach could reduce the number of paths dramatically; for some of the subjects, the reduction was several orders of magnitude.

Our second study is a user study in which we investigated the usefulness of the operational abstraction in improving user productivity in debugging faulty models. This study illustrates that the operational abstraction can assist users in repairing a faulty model more efficiently and with greater confidence, then when performing the same task without the assistance of the abstraction.

In the final study, we evaluated the feasibility of using the operation for two metamodel management and evolution tasks: measuring metamodel coverage, and refining input metamodels by annotating them with transform-specific information. The results show that, for the subjects considered, metamodel coverage could be computed accurately and useful information could be added to enrich the metamodels.

Contributions

Our technique computes sound and complete abstractions of model transforms. Moreover, it does this efficiently by performing selective path exploration. The inferred transform abstractions that can serve as valuable aid to transform users in performing tasks such as constructing a model, debugging a faulty model, or enriching a metamodel.

The contributions of our work are

- The definition of an operational abstraction of model transforms that captures essential transform semantics from the user perspective
- The description of a technique that performs selective path enumeration, dynamic path exploration, and trace synthesis to construct abstractions
- The discussion of different applications of transform operational abstractions
- The implementation of the technique for XSLT-based model-to-text transforms
- The presentation of empirical results, which illustrate the effectiveness of selective path enumeration and the usefulness of the abstraction for two applications.

2 Transform Abstraction

Intuitively, the operational abstraction illustrates the transform output, and captures all influences of input-model entities on the output. An input-model entity may contribute to an output fragment by being propagated to the output, or it may control the generation of an output fragment. The abstraction also retains the relevant structure of the transform, while removing irrelevant parts.

Definition 1 (Transform abstraction) *The* transform abstraction *is composed of one or more output fragments and is defined by the following grammar*

$$
\begin{array}{lll}
\mathcal{T} & ::= & \mathcal{T}\ O_F \mid O_F \\
O_F & ::= & \psi[O_F]^* \mid G[O_F] \mid O_F \vee O_F \mid T_F \\
T_F & ::= & l \mid \psi \mid T_F\ l \mid T_F\ \psi \\
G & ::= & \psi\ \approx\ l \mid \psi\ \approx\ \psi \mid G \wedge G \\
\approx & ::= & = \mid \neq \mid < \mid \leq \mid > \mid \geq
\end{array}
$$

where O_F is an output fragment, T_F is a text fragment, G is a relational expression (or guard), ψ is a metamodel access path, and l is a string literal.

A *metamodel access path* is a path in the model starting at the root element and terminating at an element or attribute. For example, `userData.user` and `userData.user.middle` are access paths in the input model of `PrintUser`. The vocabulary of a transform is restricted to string literals (that can appear in the output) and input-model entities (represented as access paths).

Figure 3 presents the operational abstraction of `PrintUser`. As can be seen, the abstraction contains the literals (enclosed in quotes) and input-model entities (shown as access paths) that can occur in the output. Moreover, the abstraction has repeating fragments (shown as $\psi[\ldots]^*$) and conditional fragments (shown in the guarded notation $G[\ldots]$). Such an abstraction captures the essence of the transformation logic, thereby, making it easy for the transform user to understand the logic. For example, the user can see that the input-model attribute `user.middle` occurs in the output if the attribute `user.hasMiddle` is set to true (line 2). Similarly, the abstraction shows that the transform prints information about zero or more children elements (lines 6–12).

```
1.  ψ1[
2.      "--- User ---\n"
3.      user.first"" G1[user.middle""]user.last"\n"
4.      G2[
5.          "--- Children ---\n"
6.          ψ2[
7.              user.children.user.first
8.                  G3[user.children.user.middle]
9.                  user.children.user.last"\n"
10.             G4["Boy "user.children.user.age] ∨
11.             ¬G4 ∧ G5["Girl "user.children.users.age]"\n"
12.         ]*
13.         G6["User has children\n"]
14.     ]
15. ]*
    ψ1 : userData
    G1 : userData.user.hasMiddle = true
    G2 : userData.user.children ≠ null
    ψ2 : userData.user.children
    G3 : userData.user.children.user.hasMiddle = "true"
    G4 : userData.user.children.user.gender = "M"
    G5 : userData.user.children.user.gender = "F"
    G6 : userData.user.children.user.length > 0
```

Figure 3: The operational abstraction for `PrintUser`. (For brevity, the access-path prefix `userData` is excluded in the abstraction.)

Moreover, note that the internal implementation details of the transform are abstracted away. For instance, the transform code tracks the number of children of a user in the internal variable `totalChildren`, and uses the variable to create a conditional output fragment (line 17 in Figure 1). This implementation detail is irrelevant from the user's perspective. Instead, the pertinent information for the user is that the conditional fragment "User has children" is generated based on the number of `<children>` elements in the input. This is precisely represented in line 13 of the abstraction.

The abstraction captures the transform structure. In that sense, it is a projection of the transform, created after propagation of data from the input to the input. As mentioned earlier, we intend the transform to be sound and complete. Our abstraction-synthesis technique, which we present next, creates such abstractions.

3 Synthesizing the Abstraction

Our approach constructs the abstraction by (1) enumerating the transform paths to be analyzed, (2) analyzing the paths dynamically to generate path-specific execution traces, and (3) composing together the traces, along with some simplification.

To facilitate the subsequent discussion, we introduce definitions. An *input statement* is a statement that reads an input-model entity e. An *output statement* is a statement that writes an entity of the output; the write can occur to a persistent output file or an in-memory output buffer. A statement can be both an input statement and an output statement. A *data dependence* is a triple (s_d, s_u, v) where s_d is a statement that defines variable v, s_u uses v, and there exists a path from s_d to s_u along which v is not redefined. A statement s is *control dependent* on branch (s_p, L) if there exists two (or more) branches out of s_p such that following the branch labeled 'L' causes s to be reached definitely, whereas following another branch, s may not be reached.

3.1 Path Enumeration

To generate a complete abstraction, all acyclic paths through the transform must be explored. Because this can be prohibitively expensive, we developed a *selective* path-enumeration technique that also guarantees that the abstraction

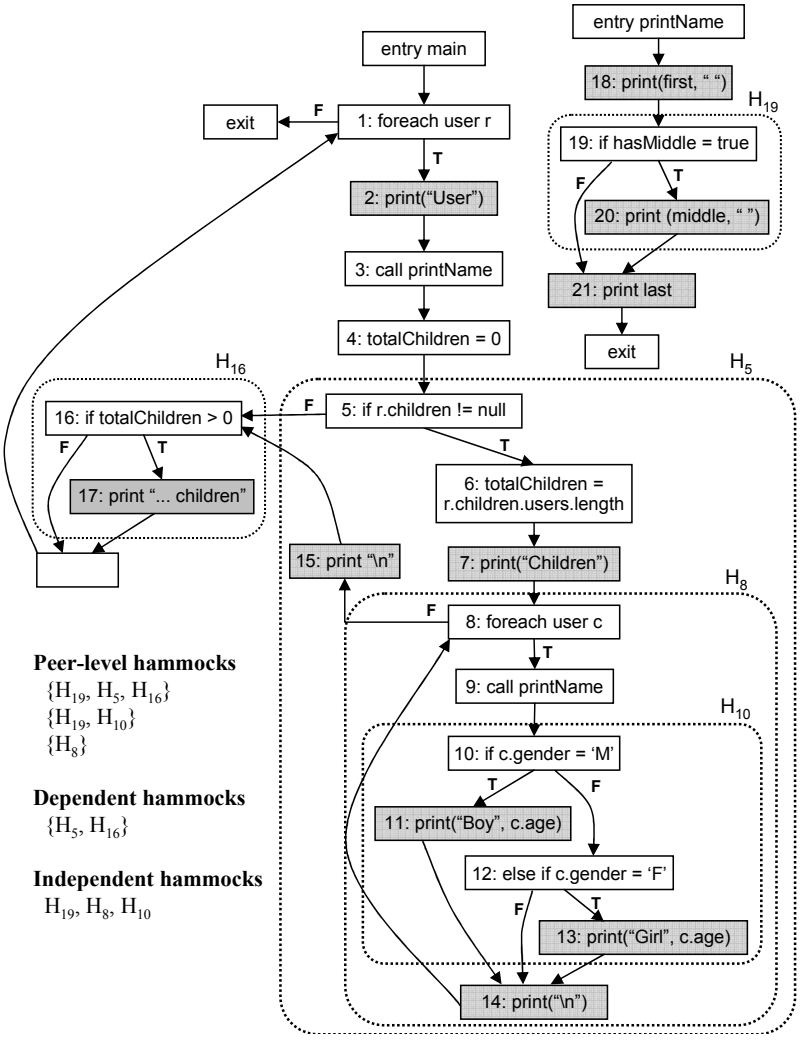

Figure 4: Control-flow graphs for `PrintUser`. Five hammocks, labeled with the hammock entry nodes, in the the CFGs are highlighted to illustrate hammock classification. The shaded nodes represent the output statements.

---

is complete. Our technique is based on the observation that the full combination of paths induced by some sequence of conditional statements is unnecessary.

We formally characterize selective path exploration based on dependence chains that start at an input-model entity and terminate at an output statement.

Definition 2 (Dependence chain) *A* dependence chain *is a sequence of statements* $\langle s_1, s_2, \ldots, s_k \rangle, (k > 1)$, *such that* s_1 *is an input statement;* s_k *is an output statement; for* $1 \leq i \leq k-2$, s_{i+1} *is data dependent on* s_i*; and either* s_k *is data dependent on* s_{k-1} *or* s_k *is control dependent on* s_{k-1}.

For example, in `PrintUser`, $\langle 8, 16, 17 \rangle$ is a dependence chain: statement 8 is an input statement, statement 17 is an output statement, statement 16 is data dependent on statement 8, and statement 17 is control dependent on statement 16.

The combinatorial blowup in the number of paths occurs in the presence of sequences of conditional statements. We define such sequences in terms of hammock graphs and peer-level hammocks.

Definition 3 (Hammock graph) *A* hammock graph H *is a sub-graph of a control-flow graph* G *with a unique* entry node $e \in H$ *and a unique* exit node $x \notin H$ *such that (1) all edges from* $(G - H)$ *to* H *go to* e*, and (2) all edges from* H *to* $(G - H)$ *go to* x *[8].*

To illustrate, Figure 4 displays the control-flow graphs for the functions in `PrintUser`. A *control-flow graph* (CFG) for a function contains nodes that represent statements and edges that represent the flow of control between statements. In the figure, five hammock graphs ($H_5, H_8, H_{10}, H_{16}, H_{19}$) are highlighted; the subscript of a hammock represents the hammock entry node. For example, for H_{19}, conditional node 19 is the entry node and node 21 is the exit node.[1]

Definition 4 (Peer-level hammocks) *Let* $\{H_1, H_2, \ldots, H_k\}$*,* $(k > 1)$*, be a set of hammocks with entry nodes* $e_1, e_2, \ldots, e_k$*. The hammocks in the set are* peer-level hammocks *if and only if the* $e_i, (1 \leq i \leq k)$*, have the same control dependence.*

In Figure 4, H_5 and H_{16} are peer-level hammocks because their entry nodes (nodes 5 and 16) have the same control dependence: $(1, T)$. Peer-level hammocks can occur in different functions. To identify such peers, interprocedural control dependences must be computed. One of the sources of interprocedural control dependence is call relations, which cause a statement in a called function to be control dependent on statements in the calling functions [22]. In `PrintUser`, statement 19 is control dependent on $(1, T)$, through call site 3, and on $(8, T)$ through call site 9. Because statement 19 (the entry node of hammock H_{19}) has the same control dependence as the entry nodes of H_5 and H_{16}, it is a peer of those hammocks. Similarly, H_{19} and H_{10} are peer-level hammocks because of the common control dependence $(8, T)$.

The presence of peer-level hammocks causes the exponential number of paths to be analyzed. But, not all peer-level hammocks have to be analyzed completely: some hammocks can be omitted because they do not influence the output; also, some hammocks can be traversed independently of their peers, thereby, avoiding the exponential blowup in the analyzed paths.

Definition 5 (Irrelevant hammock) *An hammock* H *is an* irrelevant hammock *if and only if* H *contains no output statement and* H *has no incoming or outgoing dependence chain.*

Definition 6 (Independent hammock) *A hammock* H *is an* independent hammock *if and only if* H *contains an output statement and* H *has no incoming or outgoing dependence chain.*

An irrelevant hammock has no influence on the output, either directly or indirectly. Therefore, paths through such a hammock need not be explored. An independent hammock influences output directly because it contains output statements, so paths in the hammock have to be explored. However, because such a hammock has no incoming/outgoing dependences, it need not be explored in combination with its peers—it can be explored *separately* from its peers. In Figure 4, H_8, H_{10}, and H_{19} are independent hammocks, which can be explored independently from their peers. There are no irrelevant hammocks in Figure 4.

Definition 7 (Dependent hammock) *A hammock* H *is a* dependent hammock *if and only if* H *has an incoming or outgoing dependence chain.*

[1] In the presence of exception-handling constructs, a hammock can span multiple procedures—*i.e.,* the hammock entry and exit nodes can occur in different procedures—which can complicate analysis. Our approach currently does not handle such cases. None of the subjects in our empirical studies contained such hammocks.

In Figure 4, H_5 and H_{16} are dependent hammocks because there is a data dependence, via variable `totalChildren`, between them. Dependent hammocks can require the traversal of combinations of paths. However, the exploration of all paths—in this case, 16 paths—may not be optimal. Because H_5 contains independent hammocks H_8, H_{10}, H_{19}, which would be explored separately, they are irrelevant for the combined traversal of H_5 and H_{16}. Instead of 16 paths, only four paths (two each in H_5 and H_{16}) need to be explored. Thus, independent hammocks also have the property that they are irrelevant with respect to their containing hammocks.

For `PrintUser`, exhaustive enumeration would result in 33 paths, whereas selective enumeration results in 14 paths only: four paths through H_{19} via the two calls to `printName`, two paths in H_8, three paths in H_{10}, four paths in the combined exploration of H_5 and H_{16}, and one path along branch $(1, F)$.

The path-enumeration algorithm consists of three steps.

Step 1 In this step, the algorithm performs two tasks. First, it identifies all hammocks in the transform, by analyzing each function. Second, it computes the set D of dependence chains. For each output statement s, it computes backward transitive data dependences to identify the chain $\langle s_1, s_2, \ldots, s\rangle$. If s_1 is an input statement, it adds the chain to D. Then, it identifies the control dependence of s, say s_p, and computes backward transitive data dependences for s_p: $\langle s_1, s_2, \ldots, s_p\rangle$. If s_1 is an input statement, it adds the chain to D.

Step 2 In the second step, the algorithm classifies hammocks as irrelevant or independent. If a hammock H contains no output statement and no statement in H appears in any dependence chain in D, H is classified as an irrelevant hammock. If H contains an output statement and no statement in H appears in any dependence chain in D, H is classified as an independent hammock. The remaining hammocks are dependent hammocks whose peer sequences need to be computed.

Step 3 In the final step, the algorithm iteratively identifies peer-level hammocks and selectively expands the paths. First, it computes intraprocedural paths for each method. Then, it expands the intraprocedural paths by iteratively inlining the paths in the called methods (ignoring calls to external, or library, methods) until all calls have been expanded. The algorithm handles recursion to ensure that an interprocedural cycle induced by recursive calls is traversed only once.

A path is represented as a sequence of branches, hammocks, and calls. Thus, the initial path in function `main()` is $\pi_1 = ((1, T), 3, H_5, H_{16})$. In the second iteration, the algorithm expands call node 3, which results in $\pi_2 = ((1, T), H_{19}, H_5, H_{16})$. Because there are no call nodes in the path, the algorithm has identified H_{19}, H_5, and H_{16} as peer-level hammocks.

The algorithm expands each of these hammocks to discover additional paths, call sites, and hammocks. H_{19} is an independent hammock; therefore, it is expanded separately, resulting in the computation of two paths through that hammock. Because H_5 and H_{16} are dependent, they must be expanded together. The expansion of H_{16} results in two paths. The expansion of H_5 results in paths $\pi_3 = ((5, T), H_8)$ and $\pi_4 = ((5, F))$. Because H_8 is an independent hammock, it is irrelevant with respect to H_5 (the containing hammock) and can be expanded separately. Thus, the expansion of H_5 completes with the identification of π_3 and π_4; together with its dependent peer, H_{16}, four path combinations are identified. The expansion of H_8 proceeds in a similar manner.

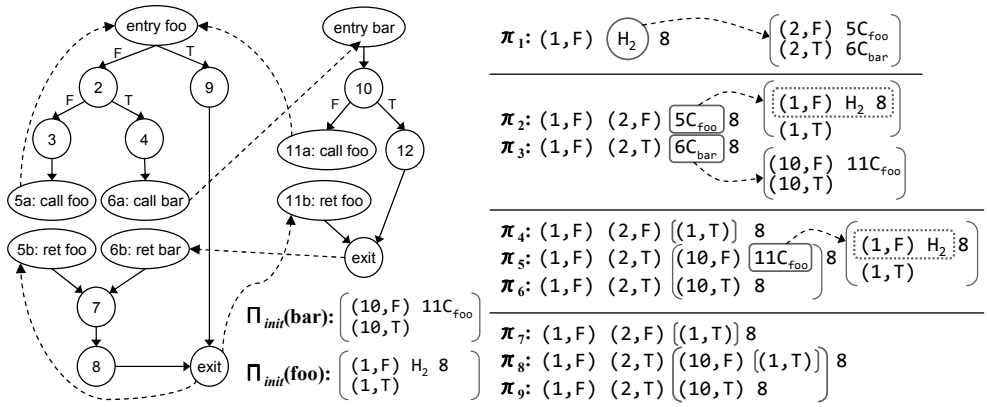

Figure 5: Illustration of path enumeration in the presence of recursion.

Path Enumeration in the Presence of Recurison

In the presence of recursion, path expansion is controlled to ensure that cyclic subpaths do not occur in the enumerated paths. Figure 5 presents an example to illustrate the analysis of recursion. It shows an *interprocedural control-flow graph*, in which a call site is represented using a *call node* and a *return node*. At each call site, a *call edge* connects the call node to the entry node of the CFG of the called function; a *return edge* connects the exit node of the CFG of the called function to the return node. For example, call site 6 in function `foo()`, which calls `bar()`, is represented with call node 6a and return node 6b; these nodes are connected to the entry node and the exit node, respectively, of the CFG of `bar()`. The example contains two instances of recursion: a direct recursion at call site 5, and an indirect recursion via call sites 6 and 11.

$\Pi_{init}(\texttt{foo})$ and $\Pi_{init}(\texttt{bar})$ contain the intraprocedural paths, with hammock nodes unexpanded, for the two methods. The right side of the figure illustrates the path expansion for path $((1, F), H_2, 8)$, which occurs in three iterations. In the first iteration, hammock node H_2 is expanded to two paths: $((2, F), 5C_{foo})$ and $((2, T), 6C_{bar})$. In the second iteration, path $\pi_2 : ((1, F), (2, F), 5C_{foo}, 8)$ is processed, where $5C_{foo}$ can potentially be expanded to the two paths in $\Pi_{init}(\texttt{foo})$. The first of the two paths would result in a cycle caused by branch $(1, F)$; therefore, that path is ignored, and π_2 is expanded to π_4. Similarly, call node $6C_{bar}$ in π_3 is expanded to paths π_5 and π_6 in the second iteration. In the final iteration, call node 11C_{foo} in π_5 is considered for expansion, and the first candidate, $((1, F), H_2, 8)$, is discarded. After this iteration, no call sites occur in any of the paths; therefore, the path expansion terminates with the final set of path $\{\pi_7, \pi_8, \pi_9\}$.

3.2 Path Exploration

The second phase consists of a dynamic analysis, which explores transform paths by forcing execution to proceed along a desired path and collects runtime information. We call the technique *forced path exploration* (FPE). FPE works by modifying the outcome of a conditional statement, at runtime, to guide the execution along a specific branch from the conditional.

To traverse a particular path, the transform is instrumented and executed on an empty input model. The path to be explored is represented as a sequence $\pi = (n_1, n_2, \ldots, n_k)$, where the n_i represent either branches or hammocks. The execution proceeds normally until it reaches n_1. After that, execution proceeds either normally or in a forced

```
 1. <DataTaint select="first">
 2.   <taint parentNode="user" currentNode="user.first"/>
 3. </DataTaint>
 4. <DataTaint literal=""/>
 5. <ControlTaint stmt="19" branch="T"
 6.  op="==" rValue="true">
 7.    <left select="hasMiddle"><taint parentNode="user"
 8.     currentNode="NULL" /></left>
 9.    <DataTaint select="middle">
10.      <taint parentNode="user" currentNode="user.middle"/>
11.    </DataTaint>
12.    <DataTaint literal=""/>
13. </ControlTaint>
14. <DataTaint select="last">
15.   <taint parentNode="user" currentNode="user.last"/>
16. </DataTaint>
```

Figure 6: Sample execution trace generated by dynamic taint analysis along the transform subpath $(18, 19, 20, 21)$.

manner depending on the type of n_i. If $n_i = (s_p, L)$ is a branch, the instrumentation probes discard the outcome of s_p and force the traversal along the branch labeled L. If $n_i = H$ is a hammock, the instrumentation probes let execution proceed normally through H: *i.e.,* the normal outcome of the entry node of H determines the traversed branch. Because that H is an irrelevant hammock, no specific path needs to be taken through it. In this manner, a forced execution consists of *interleaved* normal and coerced branch outcomes.

In addition to the probes for altering branch outcomes, the technique adds probes for collecting runtime information. The probes capture all accesses of input-model entities and generation of output, and flow across data and control dependences, in the manner of dynamic taint analysis [5]. This analysis is presented in our previous work [6, 15]. Briefly, the probes generate trace information at each assignment, conditional, and output statement. A key feature of the dynamic analysis is that it distinguishes different types of taint marks: data taints, control taints, and loop taints. A *data taint* is propagated at assignments and statements that directly, or indirectly, construct an output fragment. A *control taint* is propagated at conditional statements to the output fragments constructed in the scope of the conditional statements. A *loop taint* is propagated, in a similar manner, at looping constructs.

Figure 6 shows the trace generated for path $(18, 19, 20, 21)$ in function `printName()` when `printName()` is invoked from call site 3 in `main()` (Figure 1). In the trace, a `<DataTaint>` element captures information about the input-model entities that are propagated to output statements via assignments. Thus, lines 1–3 of the trace record the output of `userData.user.first` in line 18 of `printName()`; similarly, lines 12–14 of the trace record the printing of `userData.user.last` in line 21. A `<ControlTaint>` element captures information about the predicate in a conditional statement: the left-hand and right-hand expressions (which could represent an input-model entity or a string literal), and the comparison operator. Lines 4–11 capture information about statements 19–20 in `printName()`. Note that the nested `<DataTaint>` element (trace lines 8–10) illustrates that `userData.user.last` is guarded by the condition in statement 19.

During forced exploration, the transform executes on an empty input model. Therefore, it attempts to access non-existing input-model elements, which can cause the runtime environment to raise null-pointer exceptions. To handle such cases, appropriate instrumentation that intercept (*e.g.,* via aspect weaving) references to input-model elements and construct empty objects on-the-fly could be done. Our current implementation (Section 5.1), which analyzes XSLT-based transforms, delegates this requirement to the underlying XSLT framework, which automatically returns empty objects for non-existing input-model elements.

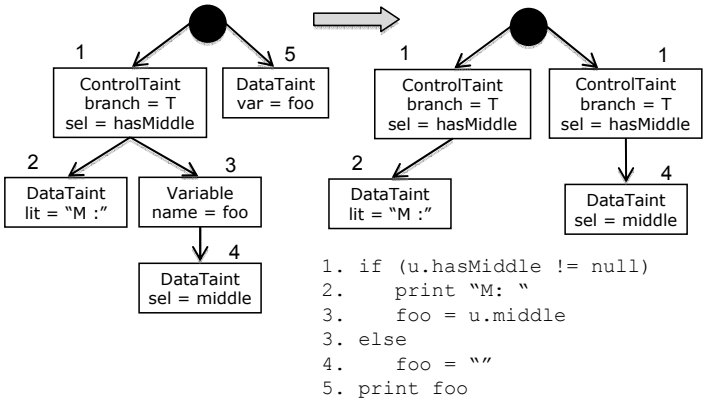

Figure 7: Illustration of variable propagation.

3.3 Abstraction Synthesis

The final phase of our approach combines the individual execution traces to synthesize the operational abstraction. The algorithm first merges the individual execution traces, and then converts the merged trace to the abstraction.

Step 1: Trace Merging To merge the individual traces, Step 1 preprocesses the traces to make them suitable for merging. First, this step removes from the trace each control-taint subtree that does not correspond to a coerced branch outcome. Recall that FPE involves interleaved normal and coerced branch evaluations. Normal execution occurs through a hammock H whose exploration is not pertinent for the particular path. Thus, information captured in the trace within the scope of conditionals that evaluated normally is irrelevant and, consequently, removed.

Second, this step resolves trace entries where variables occur because the vocabulary of the abstraction is restricted to contain access paths and literals only. Intuitively, in this step, the access path ψ at an assignment $v = \psi$ is propagated forward to the uses of v. To illustrate, consider the code fragment and the trace representations shown in Figure 7. The trace on the left is the one generated after the execution of the true branch of the `if` condition in line 1. Node 3 is a `<variable>` node, which represents the variable assignment in line 3. Node 4 is a `<DataTaint>` node representing the access path `u.middle`. Node 5 is a `<DataTaint>` node for the output in line 5. The algorithm replaces node 5 with the path $(1, 4)$: the modified trace represents the case that `u.middle` is printed at line 5 under the condition $(1, T)$. In general, the path that replaces a variable-use node n_{vu} includes the `<DataTaint>` node n_{dt} (node 4 in Figure 7) and all control-taint ancestors of n_{dt} until the least common ancestor of n_{dt} and n_{vu}. The algorithm performs the propagation transitively.

After the traces have been preprocessed, merging is performed as a simple tree-merge operation, as illustrated in Figure 8. The tree on the left shows the trace fragment generated for the path traversed along branches $(1, T)$, $(5, T)$, $(8, T)$, and $(10, T)$. Similarly, the tree in the center is the trace fragment for the path traversed along $(1, T)$, $(5, T)$, $(8, T)$, $(10, F)$, and $(12, T)$. The tree on the right results from merging these two trees. The merged trace captures the executions of both the paths.

Step 2: Abstraction Construction The algorithm for creating the operational abstraction operates on the merged trace. The algorithm traverses the merged trace in a breadth-first manner and processes each node based on its type (using the following rules) to generate parts of the abstraction.

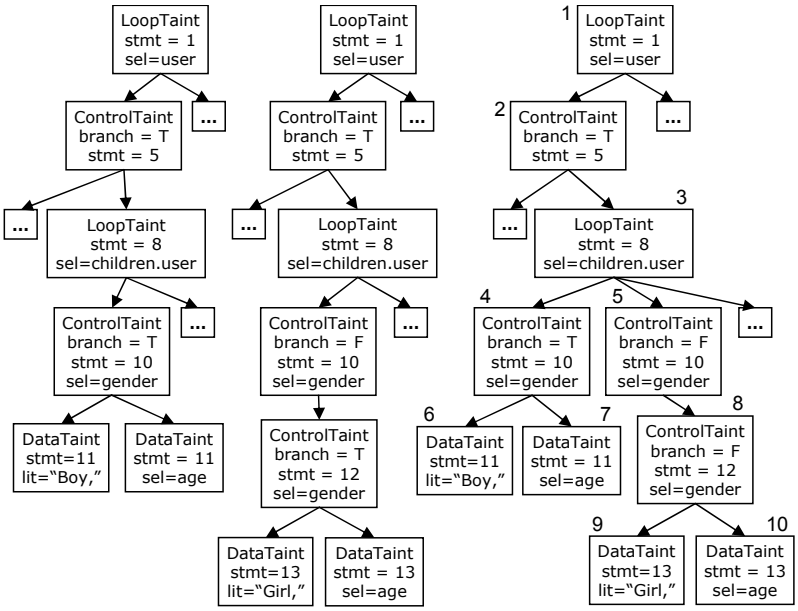

Figure 8: Illustration of trace merging: the traces on the left and in the center are merged to yield the trace on the right.

---

1. For a data taint, an access path (ψ) or a literal (l) is created in the abstraction
2. For a control taint, a guard ($G[\ldots]$) is created from the relational expression captured in the taint information
3. For a loop taint, a repeating fragment ($\psi[\ldots]^*$) is created
4. For a path of consecutive control taints, a conjunction of guards ($G_1 \wedge G_2$) is created
5. For two sibling control taints for the same statement, but different branches, alternative output fragments ($O_F \vee O_F$) are created

Additionally, to create the access paths, the algorithm propagates the `select` values from loop taint nodes to all descendant nodes.

To illustrate, consider the merged tree shown on the right in Figure 8. The algorithm traverses the tree starting at node 1, where it creates a repeating fragment $\psi_1[\ldots]^*$ in the output, where $\psi_1 =$ `user`. Next, at node 2, it creates the guard G_1 : `user.children` $\neq$ `null`. To create the complete access path, the algorithm propagates the `select` value from node 1 (a loop node) to node 2. Next, at node 3, a nested repeating fragment $\psi_2[\ldots]^*$ is created. At this point, the abstraction is

$$\psi_1[G_1[\psi_2[\ldots]^*]]^*$$

On encountering the sibling control taint nodes 4 and 5, the algorithm creates two alternative guarded fragments: $G_2[\ldots] \vee G_3[\ldots]$, where G_2 : `user.children.user.gender` $=$ `"M"` and G_3 : `user.children.user.gender` $\neq$ `"M"`. At node 6, $G_2[\ldots]$ is expanded to G_2[`"Boy,"`] and, at node 7, the fragment is recomputed as G_2[`"Boy," user.children.user.age`]. At node 8, the conjunction of G_3 and a new guard G_4 is created. After all the nodes have been processed, the computed abstraction is

$$\psi_1[G_1[\psi_2[G_2[\texttt{"Boy,"}\ \psi_3] \vee (G_3 \wedge G_4)[\texttt{"Girl,"}\ \psi_4]]^*]]^*$$

where

ψ_1 : `user`
ψ_2 : `user.children`

G_1 : `user.children` $\neq$ `null`
G_2 : `user.children.user.gender` $=$ `"M"`
G_3 : `user.children.user.gender` $\neq$ `"M"`
G_4 : `user.children.user.gender` $=$ `"F"`
$\psi_3 = \psi_4 =$ `user.children.user.age`

The complete abstraction of `PrintUser` is shown in Figure 3.

3.4 Discussion

Next, we discuss a few aspects of our technique and the trade-offs with potential (alternative) static-analysis-based techniques.

A pertinent question about the abstraction is related to the case where a transform internal variable—that has no dependence on input-model entities (*i.e.,* the value of the variable is computed internally and cannot be manipulated via the input)—can influence the transform output. Our definition of the abstraction restricts the abstraction vocabulary to access paths and literals. Although internal variables do not occur in the abstraction, their influence on the output is captured. Consider the following code fragment and abstraction

```
int i = 0                        G1[1" "user.name]
while( i < 10) {                 G1: 0 < 10
    i++;
    print i + " ";
    print user[i].name;
}
```

The abstraction provides the user with information that an integer can be printed preceding `user.name`, and that the repeating fragment occurs 10 times.

Our approach is dynamic-analysis-based; one could consider a static-analysis techniques for abstraction synthesis that are based on control-flow analysis, data-flow analysis and program-projection techniques. Our choice of dynamic analysis is based on the pragmatic consideration of wide applicability. Given the myriad of transformation implementation of technologies, a separate static-analysis infrastructure would have to be developed for each technology. Most of the popular implementation technologies (*e.g.,* XSLT, JET,[2] Velocity,[3] WebObjects[4], Kermata,[5]and Acceleo[6]) support transform execution within a Java Virtual Machine, via translation to Java. Static analysis of the generated Java code is not feasible because references to input-model elements are obfuscated in the bytecode by calls to API methods. Dynamic analysis requires Java instrumentation only (*e.g.,* via aspect weaving and bytecode rewriting), which involves much *less effort* and is *easily portable* across technologies, than developing static-analysis tools.

Our choice of forced execution is based on the simplicity of the technique in that it avoids the (expensive) generation of inputs to cover specific paths. Although forced execution introduces unsoundness in dynamic analysis, it does not cause unsoundness in the abstraction in the sense that the abstraction contains no path that is not already present in the original transform.

4 Applications of the Abstraction

The abstraction, synthesized by our technique, can be useful for different applications; we briefly discuss some of these.

[2] `http://www.eclipse.org/modeling/m2t/?project=jet`
[3] `http://velocity.apache.org`
[4] `http://www.apple.com/ca/WebObjects`
[5] `http://www.kermeta.org`
[6] `http://www.acceleo.org/pages/home/en`

4.1 Transformation Comprehension

By capturing the structure of the output and the influence of the input on the output, the abstraction inherently supports the transform user's perspective, which is focused on inputs and outputs. The user can browse the abstraction to perform efficiently tasks that require comprehension. For instance, the user can construct appropriate input models by consulting the abstraction. The `PrintUser` abstraction shows that the presence of attribute `middle` is not sufficient for it to be propagated to the output—attribute `hasMiddle` must also be present, specifically with value `true`. Such information can guide the user in creating proper input models.

Another task involves debugging a faulty input model for which the transform does not generate the expected output. The abstraction can assist the user in efficiently debugging a faulty model. To investigate this hypothesis, we conducted a user study, which is described in Section 5.3.

4.2 Metamodel Coverage and Transform Footprints

Metamodel coverage [25] identifies parts of the input metamodel that are touched, or accessed, by a transform. The coverage measure is useful for identifying the scope of a transform. It lets users assess the relevance of a transform for their metamodels, and whether the transform serves their transformation needs. Because the operational abstraction captures all input-model entities that affect the output, it can be used to compute the metamodel coverage in a straightforward manner. (In the case where an input-model entity is accessed, but does not influence the output in any way, the abstraction would not capture the entity, which would cause the coverage to be under-approximated.)

Coverage can also be computed for an input-model instance instead of the input metamodel. Jeanneret, Glinz, and Baudry [12] present techniques for computing the metamodel and model footprint of a model operation. Our operational abstraction can be used to estimate the footprint of a model instance as well.

4.3 Metamodel Refinement

The operational abstraction can be used to perform *transform-specific* metamodel refinement. *Refinement* enriches a metamodel with transform-specific information that is missing in the metamodel. The metamodel can be augmented in three ways, two of which enrich attributes and one enriches elements.

First, domain information about attributes can be added to the metamodel. The guards in the abstraction capture comparisons of input-model attributes with literals. By analyzing such guards, we can identify relevant values from the domains of the attributes and relevant sub-domains. To illustrate, consider the `PrintUser` abstraction shown in Figure 3. We can infer from the guards G_3 and G_4 in the abstraction that `"M"` and `"F"` are two relevant values in the (infinite) domain of `userDara.user.children.user.gender`. This information can be added to the metamodel to enrich it.

Second, attributes can be annotated to indicate whether they are "optional" or "mandatory." For instance, line 3 of the `PrintUser` abstraction (Figure 3) shows that `user.first` and `user.last` are unguarded, whereas `user.middle` is guarded. Therefore, in the metamodel, `user.middle` can be marked as optional, and `user.first` and `user.last` can be annotated as mandatory.

Third, elements can be annotated with cardinality information. In the case of `PrintUser`, the input metamodel can be augmented to show that `userData.user` and `userData.user.children.user` can occur zero or more times in any instance of the metamodel.

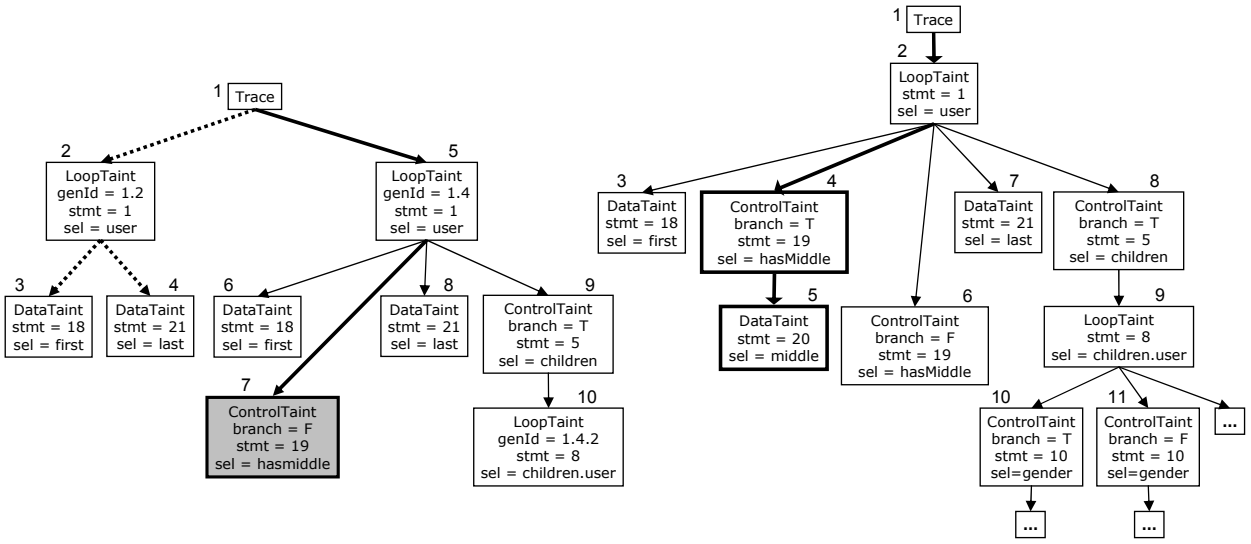

Figure 9: Illustration of fault repair. The execution trace on the left is generated for the input model shown in Figure 2(a). The trace on the right is the merged trace (shown partially) constructed by Step 1 of abstraction synthesis (Section 3.3).

4.4 Critical Metamodel Regions

Previous research has presented the notions of critical and forgiving input regions in the context of applications that process complex inputs [4]. A *forgiving input region* is one where small changes induce correspondingly small changes in the behavior and output of the application. A *critical input region* is one where small changes can cause the application behavior and output to change dramatically. Although these concepts have not been applied in the domain of model transforms, they intuitively make sense for model transforms and can be very useful for tasks, such as managing and evolving metamodels.

The transform abstraction captures the influence of input-model entities on the output. Therefore, it can be leveraged to identify critical and forgiving metamodel regions, based on quantification of influence. The quantification could be computed from the extent of the output influenced by an input. More interestingly, the *nature of influence* would be important to consider: an input-model entity could affect only a small part of the output, but one that is critical for the output to conform to a metamodel; conversely, an input could affect large parts of the output that have no relevance for metamodel conformance. There are other interesting research aspects of the problem as well. For instance, critical regions for a metamodel could be computed from a set of transforms (that use the same input metamodel) or for chained transformations (where the output of one transform is the input to another transform).

4.5 Automated Fault Repair

The operational abstraction can be leveraged to improve the automated fault-repair technique that we developed in previous work [15]. That technique computes repair actions to fix a faulty input model by analyzing the execution trace generated during the execution of a transform on the faulty model. The operational abstraction can enable the technique to compute more accurate repair actions.

A *recommendation* is a set of repair actions and a *repair action* is an atomic change that can be made to a faulty input model. There are three types of repair actions: (1) add an attribute, with an arbitrary value, to an element (specified as an access path), (2) add an element to another element, and (3) set the value of an attribute.

To illustrate fault repair, suppose that the input model in Figure 2 contained a fault: missing attribute `hasMiddle` (with value `true`) for the second `<user>` element. This fault causes attribute `middle` to not be printed. Starting from this symptom of the fault, the user attempts to diagnose the fault by examining the input model in an incremental manner, guided by the hierarchy of taint marks enclosing the incorrect output fragment [6]. In this process, the user identifies the *fault index*, which is the innermost enclosing taint mark associated with the faulty input-model element/attribute. In this example, the incorrect output fragment is a missing string, which is enclosed by the control taint propagated at statement 19 of `PrintUser`. The trace for the faulty input model is shown on the left in Figure 9; the highlighted control-taint node is the fault index.

The fault-repair technique [15] computes repair actions by identifying correct output fragments that are "similar" to the incorrect fragment. In our example, the technique would attempt to find another user element for which some output is generated under the conditional in line 19 of `PrintUser`. Algorithmically, the technique walks up in the trace tree—starting at the fault index—to the root node. This path is called the *fault path*; highlighted in the trace in Figure 9. Next, the technique walks down from the root node to find another path that is at least as deep as the fault path and that ends at a data taint node. (We refer the reader to Reference [15] for a precise description of the trace-traversal algorithm.) In this case, there is no such path in the execution trace because none of the `<user>` elements in the input model contains both attributes `hasMiddle` (with value `true`) and `middle`. Thus, the technique would compute no repair actions.

The effectiveness of the repair technique, thus, depends on the richness of the faulty input model: that is, whether the model contains enough patterns of correct output fragments, on the basis of which repair actions can be computed. In this example, it does not contain such fragments. The technique could be made more effective if it were to analyze other "exemplar" input models as well (when available), in addition to the faulty input model. Our abstraction can, in fact, serve this purpose because it captures all output fragments that can potentially be generated. Specifically, the repair technique can analyze the merged execution trace that is generated prior to abstraction construction (see the description of Step 1 in Section 3.3). The trace of the right in Figure 9 shows (partially) the merged trace. Guided by the fault path, the repair algorithm would walk the highlighted path $(1, 2, 4, 5)$, and compute repair actions based on the trace information associated with nodes 4 and 5. (Repair actions are computed starting at nodes that are at the same depth in the trace tree as the fault index [15].) By analyzing node 4, the technique computes two repair actions

```
ADD hasMiddle TO 1.4
SET 1.4/hasMiddle TO true
```

The analysis of node 5 yields no repair action because user element `1.4` already contains attribute `middle`. However, if this attribute were missing, the technique would compute two more repair actions

```
ADD middle TO 1.4
SET 1.4/middle TO *
```

5 Empirical Evaluation

We implemented our approach for XSLT-based model transforms and conducted empirical studies to evaluate different aspects of the abstractions and the technique for constructing them. In the first study, we evaluated the effectiveness of selective path enumeration. Our second study is a user study, in which we investigated the usefulness of abstractions in improving user efficiency in debugging faulty input models. The third study focuses on two applications of abstractions for metamodel management and evolution: metamodel coverage and metamodel refinement. In the final study, we

Table 1: Subjects used in the empirical studies.

Subject	XSLT Constructs used in the Subjects						Translet Bytecode Instructions
	Templates	Loop	Condition	Data	Variables	Select	
ClsGen1	4	8	14	147	27	156	3722
ClsGen2	19	16	94	519	103	541	13270
IntGen1	2	2	5	16	9	19	938
IntGen2	6	3	40	101	15	107	3264
PluginGen	7	9	48	107	52	122	5888
Total	38	38	201	890	206	945	27082

Table 2: Effectiveness of Selective Path Exploration

Subject	All Paths	Paths with H_{Irr} Ignored	Paths with H_{ind} Optimized	Both
ClsGen1	198	110	25	23
ClsGen2	1.4 x 10^{29}	5.073 x 10^{13}	269	155
IntGen1	16	16	8	8
IntGen2	62,734	26890	62	53
PluginGen	482,879,880	19,549,215	64	50

evaluated the effectiveness of the abstraction in improving automated fault repair. After describing the experimental setup, we present the results of the studies.

5.1 Experimental Setup

We leveraged and extended the infrastructure that we had developed in our previous work [6, 15]. The implementation converts an XSLT program to a Java translet using XSLTC (`http://xml.apache.org/xalan-j/xsltc`); all the analyses operate on the Java translet. We implemented the path-enumeration algorithm using the WALA analysis infrastructure (`http://wala.sourceforge.net`). The instrumentation component rewrites the translet bytecode, using the Bytecode Engineering Library (`http://jakarta.apache.org/bcel`) to add probes for forced path exploration. The execution trace collection is implemented using a combination of bytecode rewriting and aspect weaving.

We used five XSLT transforms, listed in Table 1, as our experimental subjects. The transforms are real programs that have been developed in projects at IBM Research. Each transform takes as input a domain-specific Ecore EMF model (`http://www.eclipse.org/modeling/emf`), and generates different types of text output: `ClsGen1` and `ClsGen2` generate Java classes; `IntGen1` and `IntGen2` generate Java interfaces; and `PluginGen` generates an XML configuration file. To indicate the complexity of the subjects, Columns 2–7 list different XSLT features used in the subjects. The last column shows the number of bytecode instructions in the Java translet for each subject.

5.2 Selective Path Exploration

In the first study, we evaluated the usefulness of selective path enumeration in reducing the scope of dynamic exploration. We investigated two research questions. (1) How effective is selective path enumeration in reducing the number of explored paths over full path enumeration? (2) What are the contributions of the irrelevant-hammock and independent-hammock optimizations to the reduction?

To collect the path counts, we enumerated paths, using the different techniques, starting from the main methods. First, we computed the paths using full enumeration. Then, we applied optimizations for irrelevant and independent hammocks separately, to measure their individual contributions to the reduction. Finally, we applied both optimizations together to compute the paths for selective exploration.

Table 2 presents the data about the path reduction. Column 2 of the table lists the full paths. It shows that, for the two largest subjects (`ClsGen2` and `PluginGen`), full path exploration has to contend with exponential blowup. By ignoring

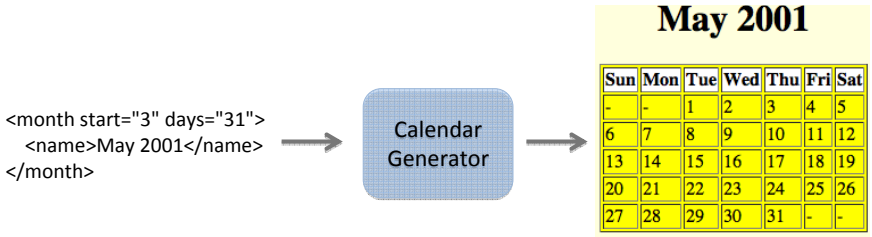

Figure 10: Sample input and output of the calendar generator transform used as the subject in the user study.

irrelevant hammocks, the number of paths decreases considerably—several orders of magnitude for `ClsGen2`—as shown in Column 3. However, the reduced number of paths are still prohibitively large.

The strategy of exploring independent hammocks separately results in a dramatic reduction in the path counts, to make path exploration practical. For `ClsGen2`, the path count is reduced by a factor of 10^{27} from 1.4×10^{29} to 269. Similarly, a remarkable reduction occurs for `PluginGen`. The last column shows that the two optimizations, when applied together, achieve better results than either optimization applied alone for four of the five subjects.

The data illustrate that selective path enumeration is indeed essential for path exploration, and that our characterization of irrelevant and independent hammocks can achieve a remarkable reduction in the number of paths to be analyzed. We note that the domain of our subjects—XSLT transforms—also plays a role the radical path reduction. Unlike general programming languages, a variable, once assigned, cannot be reassigned by some function within XSLT. This makes peer-level hammocks predominantly independent. This may not be the case for other transform implementation technologies. However, other studies [19] have reported significant path reduction (in the context of symbolic execution) in Java programs based on a similar characterization as ours.

5.3 User Study

In the user study, we investigated the following hypothesis

> A user can perform the task of identifying and fixing faults in a failure-inducing input model more efficiently when guided by the transform abstraction than without the abstraction.

Study Design

Subject Program. We used an XSLT transform called `Calendar Generator`,[7] as the subject. (Appendix A shows the source code of the transform.) The transform takes as input an XML file specifying a month, and generates as output a a visual depiction of the calendar rendered in HTML. Figure 10 shows a sample input to, and the corresponding output of, the transform.

Participant Selection. To select participants with different degrees of expertise, we identified the factors on which expertise assessment could be based. Familiarity with XSLT, XML, and HTML are the main factors, given the characteristics of the subject program. Based on this criteria, we created three participant categories: expert, typical, and novice. The *expert user* is a software engineer with experience in XSLT development. The *typical user* is a software engineer with no XSLT-development experience, but who is familiar with XML and HTML. The *novice user* has minimal XML/HTML knowledge and no knowledge of XSLT. We formed a *control group*, which would not use the operational abstraction to perform the debugging task, and an *experimental group*, which would use the abstraction.

[7] `http://incrementaldevelopment.com/xsltrick/parvez/\#calendar`

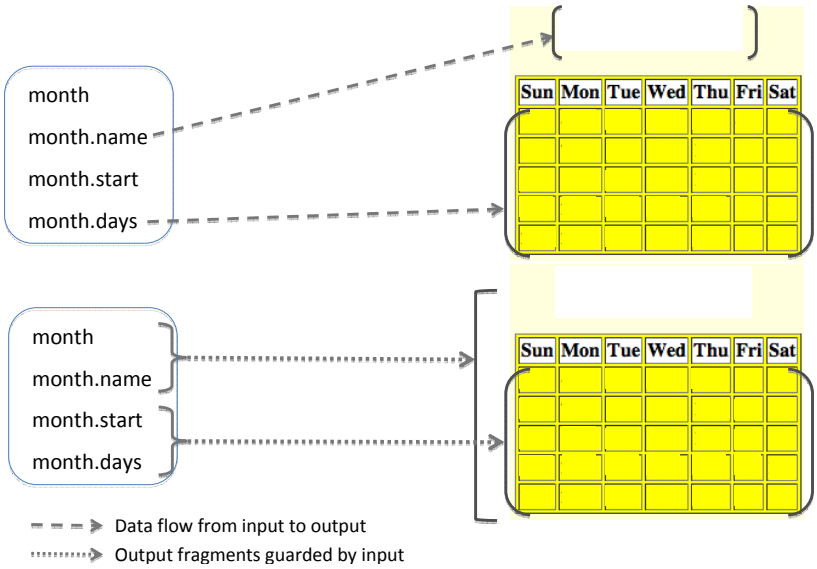

Figure 11: Visual representation of the abstraction of the calendar generator abstraction.

Each group consisted of one expert user, two typical users, and one novice user. The overall software-development experience of the users ranged from 2–10 years across all categories.

Debugging Task. We created a faulty version of the input model shown in Figure 10

```
<month week="5" name="May">
   <year>2001</year>
</month>
```

For the correct input model, the transform generates the calendar shown in Figure 10, whereas for the faulty input model, the transform generates an empty HTML page with no calendar.

We manually created the visual representation of the transform abstraction shown in Figure 11. For this study, we did not implement automated construction of the visual representation. The presentation of the abstraction via intuitive visual representation is important and, based on the output of a transform, an appropriate representation can be created in an automated manner. We leave the development of automated visualization techniques to future work.

The control group was asked to fix the model by examining the transform code, whereas the experimental group was asked to fix the model by examining the transform's operational abstraction only (and not the transform code). Both groups were also given the options of executing the transform after editing the faulty model any number of times, and examining the HTML source (in addition to the browser-rendered HTML) generated by the transform.

Measures. We used three *quantitative measures*: time taken to fix the fault (Q_n^1), number of transform executions performed (Q_n^2), and whether output HTML source is examined (Q_n^3). Our rationale behind using Q_n^2 and Q_n^3 is that they indicate the activities and effort involved in debugging. We also used three *qualitative measures*: user confidence in the correctness of the fix (Q_l^1), ease of determining the relevant elements/attributes involved in the fix (Q_l^2), and ease of selecting the value for an element/attribute (Q_l^3). Q_l^1 was measured on the scale {high, medium, low}, whereas Q_l^2 and Q_l^3 were measured on the scale {very easy, easy, somewhat difficult, difficult}.

Table 3: Results of the user study. CG refers to the control group, whereas EG refers to the experimental group.

	Time (min)		Number of Runs		Output Examined		Confidence in Fix		Element/Attribute Identification		Value Selection	
User	CG	EG	CG	EG	CG	EG	CG	EG	CG	EG	CG	EG
E1	14	6	3	3	Yes	Yes	Medium	High	V. Easy	V. Easy	S. Difficult	S. Difficult
T1	8	3	3	3	Yes	No	Medium	Medium	Easy	V. Easy	Easy	Easy
T2	7	5	3	3	Yes	No	Medium	High	S. Difficult	V. Easy	Easy	Easy
T3	11	6	5	3	Yes	No	Medium	High	V. Easy	Easy	Easy	Easy
N1	17	5	4	4	Yes	No	High	High	V. Easy	V. Easy	Difficult	Easy

Results and Analysis

Table 3 presents the results of the study. It shows the data for each of the six measures for the control group (CG) and the experimental group (EG). In all cases, a user in the experimental group took less that time than a user of corresponding expertise in the control group (Columns 2 and 3). On average, an experimental-group user took approximately 5 minutes, whereas a control-group user took over 11 minutes, to complete the task. The expert user in the control group readily delved into the transform code to get a proper understanding of the code, which the other users did not do. This explains the longer time taken by the expert than the typical group. A noteworthy aspect of the timing data is that there is much less variance among the experimental-group users than among the control-group users. This suggests that the abstraction served to *bridge the gap* between the experts and the novices—and, to some extent, made "expertise" less of a determinant of efficiency. Reducing the need for expertise is, in fact, one of the goals of productivity improvement tools, and our study seems to support the premise.

In all but one case, the experimental group did not examine the generated HTML source, whereas the control group did (Columns 6 and 7). The information in the abstraction essentially eliminated the need for examining the generated output and, thereby, made a potentially cumbersome task unnecessary.

The experimental group demonstrated higher confidence in the correctness of the fix than the control group (Columns 8 and 9). In fact, even the novice user in the experimental group showed high confidence, whereas the expert user in the control group indicated medium confidence. The lack of suitable information contributed to the weaker confidence of the control group. For instance, the control-group users were unsure whether certain attributes (*e.g.,* `week`) had to be removed from the input model or whether the attributes were referenced in the transform.

In terms of the effort involved in identifying relevant elements, attributes, and values, the experimental group rated the task "easy" or "very easy," whereas some users in the control rated this task as "difficult" or "somewhat difficult" (Columns 10–13). By explicitly showing the input-model elements and attributes used by the transform, the abstraction made this information readily available to the experimental-group users, whereas the control-group users had to wade through the transform code to dig out the information.

Although limited in scope, this study illustrates the potential value of transform abstractions in improving user productivity in performing tasks, such as debugging input models. At the conclusion of the study, we showed the abstraction for the subject to the control-group users; all the users said that, had they used the abstraction, they would have accomplished the debugging task faster.

5.4 Metamodel Management and Evolution

In this study, we evaluated the feasibility of using operational abstractions for supporting two tasks in metamodel management and evolution: metamodel coverage (Section 4.2) and metamodel refinement (Section 4.3).

Our subjects operate on a common Ecore metamodel. For each subject, we computed the coverage attained on the metamodel by analyzing the access paths in the abstractions. Specifically, we measured *attribute coverage*, *element*

Table 4: Metamodel coverage and attribute domain inference.

	Element Coverage			Attribute Coverage		
Subject	Total	Fully Covered	Partially Covered	Total	Covered	Attribute Domains
ClsGen1	20	1	4	62	15	5 (33%)
ClsGen2	20	1	4	62	24	10 (42%)
IntGen1	20	1	4	62	12	3 (25%)
IntGen2	20	1	4	62	20	7 (35%)
PluginGen	20	1	4	62	16	3 (19%)

coverage, and *partial element coverage*. A metamodel attribute is covered if its access path occurs in the abstraction. A metamodel element is covered if all of its sub-elements and attributes occur in the abstraction. An element is partially covered if some, but not all, of its sub-elements and attributes occur in the abstraction.

For metamodel refinement, we studied how the input metamodel could be enriched by adding domain information to attributes. We computed the domain information by analyzing guards that compared access paths with literals—*i.e.,* guards of the form $\psi = l$. We counted the number of attributes for which domain information could inferred in this manner.

Table 4 presents the data. Columns 2–4 show element coverage, columns 5–6 show attribute coverage, and column 7 shows the number of inferred attribute domains. The coverage data show that our subjects use a very small part of the input metamodel. Whereas element coverage is the same, attribute coverage varies across the subjects. Because Ecore is a general-purpose metamodel, sparse usage in specific contexts is to be expected. In fact, others have discussed this sparse-usage issue for general-purpose models, such as UML [12, 21, 25], which is one of the motivations behind the development of techniques such as metamodel pruning [21].

Column 7 shows the number of covered attributes for which we could infer domain information. As the data show, the abstractions could be leveraged to infer domains for a good percentage—ranging from 19% to 42%—of the covered attributes.

To summarize, this study illustrates the feasibility of leveraging transform operational abstractions for computing metamodel coverage and attribute domains, which can be useful for evolving, refining, and pruning metamodels.

5.5 Improving Automated Model Repair

The goal of the second study was to investigate how the abstraction (specifically, the merged execution trace) can improve the pattern-analysis-based model repair [15]. To do this, we compared the recommendations computed by the technique using (1) the merged execution trace generated during abstraction synthesis, and (2) the trace generated by the execution of the faulty input model only.

To generate failure-inducing input models, we used the technique of generating inputs by applying mutation operators to a valid input (*e.g.,* [7]). Specifically, we defined four mutation operators, which delete an element, delete an attribute, modify the value of an attribute based on a predefined enumeration of values, and set an attribute to empty. We applied the operators to two valid input models to generate faulty inputs. We used an upper limit of 100 for each input to bound the number of mutants. From the initial set of mutants, we eliminated mutants that caused no change in the transform output (*i.e.,* the *equivalent* mutants). Next, we removed the mutants for which fault localization did not cause the execution of incorrect paths. We performed this step because computing repair actions for failures where incorrect (resp., missing) input-model attributes are simply written (resp., missing) in the output does not require pattern-analysis-based fault repair [15]. From the set of mutants for which the same fault index was identified, we selected only one mutant. This resulted in a total of 107 faults across the five subjects.

Table 5: Comparison of recommendations calculated using (1) pattern analysis on the execution trace of the faulty input model ($PA(\mathcal{E}_{fault})$) and (2) pattern analysis using the merged execution trace generated by abstraction synthesis ($PA(\mathcal{E}_{abs})$).

Subject	Faults	Faults for which $PA(\mathcal{E}_{fault}) = PA(\mathcal{E}_{abs})$	Faults for which $PA(\mathcal{E}_{fault}) < PA(\mathcal{E}_{abs})$
ClsGen1	19	3	16 (84%)
ClsGen2	45	6	39 (87%)
IntGen1	9	2	7 (78%)
IntGen2	28	2	26 (93%)
PluginGen	6	0	6 (100%)
Total	107	13	94 (88%)

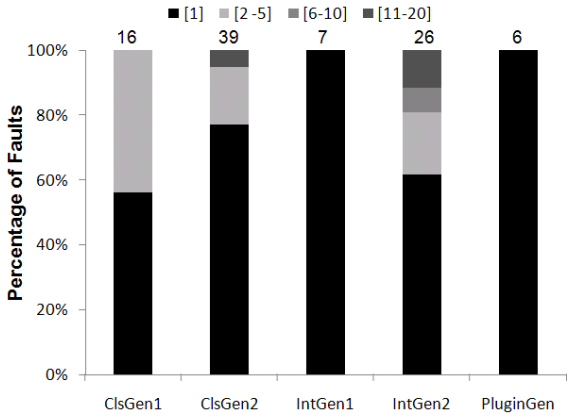

Figure 12: For the faults for which the abstraction improved pattern-analysis-based repair, the distribution of additional recommendations computed.

To collect the data, we computed, for each fault and each subject, two sets of recommendations: one set computed by the repair technique using the execution trace for the faulty model ($PA(\mathcal{E}_{fault})$), and the other set computed by the repair technique using the merged trace generated by abstraction synthesis ($PA(\mathcal{E}_{abs})$). Table 5 presents the data about the recommendations. It shows the number of faults (Column 2) for each subject, the number of faults for which $PA(\mathcal{E}_{fault}) = PA(\mathcal{E}_{abs})$ (Column 3), and number of faults for which $PA(\mathcal{E}_{fault}) < PA(\mathcal{E}_{abs})$ (Column 4). As the data show, the merged trace improved pattern analysis for a large percentage of the faults. The improvement varies from 78%, for `IntGen1`, to 100% for `PluginGen`. On average, for 88% of the faults, the merged trace improved pattern analysis.

To get insight into the magnitude of improvement, we examined the number of additional recommendations computed by $PA(\mathcal{E}_{abs})$. Figure 12 presents this data as a segmented bar chart. Each bar represents 100% of the faults for which $PA(\mathcal{E}_{fault}) < PA(\mathcal{E}_{abs})$ (Column 4 of Table 5). The segments in a bar illustrate different ranges of additional recommendations in $PA(\mathcal{E}_{abs})$. For example, consider the bar for `ClsGen2`. For 77% of the faults, $PA(\mathcal{E}_{abs})$ contained one more recommendation than $PA(\mathcal{E}_{fault})$; for 18% of the faults, $PA(\mathcal{E}_{abs})$ contained 2–5 more recommendations; for the remaining 5%, $PA(\mathcal{E}_{abs})$ contained between 11 and 20 more recommendations. For `IntGen2`, for 19% of the faults, the abstraction enabled the computation of between six and 20 additional recommendations.

Overall, the data indicate that the merged trace generated during abstraction synthesis has the potential to improve pattern-analysis-based fault repair. For our subjects, using the merged trace resulted in the computation of at least two more recommendations for 20% of the faults; for another 7% of the faults, at least six more recommendations. To verify that each additional recommendation was a valid alternative repair suggestion, we asked the transform developers to examine the recommendations. They reported that none of the suggestions were invalid. Because pattern

analysis computes alternate suggestions for fault repair, by leveraging the merged trace created during abstraction synthesis, it can offer more repair suggestions.

5.6 Threats to Validity

The most significant threats to the validity of our results are threats to external validity, which arise when the observed results cannot be generalized to other experimental setups. In our studies, we used five XSLT-based transform and, therefore, we can draw limited conclusions about how our results might hold for other transforms. Our transforms vary in complexity and generate different types of outputs (code, properties, and configuration files), which gives us some confidence that the results might apply to other XSLT transforms. For transforms implemented using other frameworks or general-purpose programming languages (*e.g.,* Java), patterns of references to input-model entities and output generation might differ, which would determine the effectiveness of selective exploration. Further experimentation with other types of transforms is required to provide evidence of the usefulness of our approach in more general settings.

Another threat to validity is the representativeness of the faulty input models. We used mutation analysis to generate the faulty models, which may not represent the types of faults that occur frequently in practice. However, based on our experience with developing transforms, we designed the mutation operators to capture commonly occurring faults. A related threat is that, in our studies, each faulty input contained a single fault.

6 Related Work

Møller, Olesen, and Schwartzbach [16] present a static flow analysis for validating XSL transforms. They use the analysis to construct an XML graph for an XSL function, which represents the possible output generated by the function. The graph consists of different types of nodes to represent elements, attributes, and text. The edges represent possible ways of composing the output XML document. Our notion of abstraction is different from the XML graph. Although the XML graph captures the output structure, it is not a projection of the transform and it does not represent guarded fragments. Our abstraction can be applied to any structured output, and is not restricted to XML documents only. Moreover, our synthesis technique is a combination of static selective path enumeration and dynamic path exploration.

Existing research has investigated other forms of program summaries, such as semantic signatures [18], output summaries (*e.g.,* [13]), and textual program summaries (*e.g.,* [11, 23]).

Qi, Nguyen, and Roychoudhury [18] define a semantic signature with respect to an output variable. The signature consists of the possible expressions for the output variable, along with the path condition under which a particular output expression is produced. Their approach computes the signature by performing symbolic execution over a relevant slice [10]. Our operational abstractions differ in that they retain the program structure, illustrate the influences of inputs on outputs, and are suitable for summarizing model transforms.

Reps, and Liblit [13] present a static-analysis approach for extracting output data formats, such as file formats, from x86 executables. The approach first constructs a hierarchical finite-state machine, which is a projection of the program control flow with respect to entry/exit/call nodes and output operations. Then, it performs value-set analysis and aggregate structure identification iteratively, to compute an over-approximation of the output values and sizes. The output format is represented as a regular expression. Our approach is dynamic, infers operational abstractions, and applies to model transforms.

Researchers have investigated techniques for selective path exploration for improving symbolic execution [19] and create semantic signatures of programs [18].

Santelices and Harrold [19] present the notion of path families, which consists of a set of paths that can be traversed as a group for symbolic execution—the individual paths in the family need not be explored separately. Our approach for selective path enumeration is similar, but simpler, in concept in the sense that further path grouping within dependent hammocks could be attempted. But, for our application, selective exploration based on the hammock characterization is sufficient. We apply this concept to the domain of model transforms, with the goal of computing abstractions. Whereas our results illustrate that selective enumeration can cause substantial path reduction for XSLT programs, the results of Santelices and Harrold show a similar trend for general Java programs.

Qi, Nguyen, and Roychoudhury [18] partition program paths to compute the program signature. Two paths are put in the same partition if they have the same relevant slice [10] with respect to an output variable. Our characterization is based on dependence chains (or static slices [26]) across peer-level hammocks. Our simpler hammock-based path enumeration is suitable for the domain of model transforms.

The underlying principle of forced exploration—altering the outcome of a predicate at run-time to force execution along a particular branch—is a general technique for observing new program behaviors without requiring the generation of new test inputs [24]. It has been investigated in different contexts, such as malware detection [17, 27], fault localization [28], and fault detection [14, 24]. Some of these approaches have also investigated techniques for overcoming the unsoundness inherent in forced path execution. For abstraction synthesis, such techniques are irrelevant as our goal is to summarize the output along static transform paths, but via dynamic exploration (for the reasons mentioned in Section 3.4). Thus, if infeasible static paths exist, our approach explores them.

An alternative approach for exploring transform paths and computing runtime information is to generate input models to attain adequate transform coverage: that is, coverage of the paths computed by our selective-enumeration technique. Existing research on model generation (*e.g.,* [2, 20]) or general test-input generation (*e.g.,* [3, 9]) could be leveraged to this end. However, in spite of the recent development of powerful automated techniques, test-input generation remains an inherently complex and expensive task. In comparison, forced exploration requires no expensive analysis (such as, constraint solving), applies to transforms and inputs of any complexity, and can easily be implemented.

7 Summary and Future Work

In this paper, we presented the notion of operational abstractions for model transforms. The operational abstraction captures the essential transformation logic, while abstracting away the implementation details, which are irrelevant from the transform user's perspective. The abstraction can assist the user in understanding the structure of the output and the influence of input-model entities on output fragments. We presented an approach that uses selective path enumeration, dynamic exploration of enumerated paths, and offline analysis and merging of execution traces to synthesize sound and complete abstractions. We discussed different applications of the abstraction, such as computing metamodel coverage, refining metamodels, and identifying critical metamodel regions. In this way, the abstraction can guide metamodel management and evolution tasks.

Our empirical results showed that, for the subjects considered, selective path enumeration substantially reduced the scope of path exploration. Our user study indicated the usefulness of abstractions in assisting users debug faulty models with greater efficiency and confidence. Finally, we empirically assessed the feasibility of leveraging abstractions for

measuring metamodel coverage and enriching metamodels with domain information.

Our evaluation was limited to XSLT-based transforms. In future work, we will investigate whether our results generalize to transforms implemented using other frameworks or implemented in general-purpose programming languages, such as Java. Our dynamic-analysis-based approach would be easier to extend to accommodate other transform-implementation technologies than techniques based on static analysis of transforms. Other potential directions for future research include investigating more applications of operational abstractions. In particular, applying the notions of critical and forgiving regions [4] to metamodels and model transforms would be interesting, and useful techniques for guiding metamodel management and evolution could be developed.

References

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

[2] E. Brottier, F. Fleurey, B. Baudry, and Y. Le Traon. Metamodel-based test generation for model transformations: An algorithm and a tool. In *Proceedings of the 17th International Symposium on Software Reliability Engineering*, pages 85–94, November 2006.

[3] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 209–224, 2008.

[4] M. Carbin and M. C. Rinard. Automatically identifying critical input regions and code in applications. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, pages 37–48, 2010.

[5] J. Clause and A. Orso. Penumbra: Automatically identifying failure-relevant inputs using dynamic tainting. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 249–259, July 2009.

[6] P. Dhoolia, S. Mani, V. S. Sinha, and S. Sinha. Debugging model-transformation failures using dynamic tainting. In *Proceedings of the 24th European Conference on Object-Oriented Programming*, pages 26–51, June 2010.

[7] T. Dinh-Trong, S. Ghosh, R. France, B. Baudry, and F. Fleury. A taxonomy of faults for UML models. In *Proceedings of the 2nd Workshop on Model Design and Validation*, October 2005.

[8] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[9] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223, 2005.

[10] T. Gyimóthy, Á. Beszédes, and I. Forgács. An efficient relevant slicing method for debugging. In *Proceedings of the 7th European Software Engineering Conference and the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 303–321, November 1999.

[11] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus. On the use of automated text summarization techniques for summarizing source code. In *Proceedings of the 2010 17th Working Conference on Reverse Engineering*, pages 35–44, 2010.

[12] C. Jeanneret, M. Glinz, and B. Baudry. Estimating footprints of model operations. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 601–610, 2011.

[13] J. Lim, T. Reps, and B. Liblit. Extracting output formats from executables. In *Proceedings of the 13th Working Conference on Reverse Engineering*, pages 167–178, 2006.

[14] S. Lu, P. Zhou, W. Liu, Y. Zhou, and J. Torrellas. PathExpander: Architectural support for increasing the path coverage of dynamic bug detection. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 38–52, December 2006.

[15] S. Mani, V. S. Sinha, P. Dhoolia, and S. Sinha. Automated support for repairing input-model faults. In *Proceedings of the 25th International Conference on Automated Software Engineering*, pages 195–204, September 2010.

[16] A. Møller, M. Ø. Olesen, and M. I. Schwartzbach. Static validation of xsl transformations. *ACM Trans. Program. Lang. Syst.*, 29, August 2007.

[17] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 231–245, 2007.

[18] D. Qi, H. D.T. Nguyen, and A. Roychoudhury. Path exploration based on symbolic output. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 278–288, 2011.

[19] R. Santelices and M. J. Harrold. Exploiting program dependencies for scalable multiple-path symbolic execution. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, pages 195–206, 2010.

[20] S. Sen, B. Baudry, and J.-M. Mottu. Automatic model generation strategies for model transformation testing. In *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations*, pages 148–164, 2009.

[21] S. Sen, N. Moha, B. Baudry, and J.-M. Jézéquel. Meta-model pruning. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, pages 32–46, 2009.

[22] S. Sinha, M. J. Harrold, and G. Rothermel. Interprocedural control dependence. *ACM Transactions on Softw. Engineering and Methodology*, 10(2):209–254, April 2001.

[23] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 43–52, 2010.

[24] P. Tsankov, W. Jin, A. Orso, and S. Sinha. Execution hijacking: Improving dynamic analysis by flying off course. In *Proceedings of the 4th IEEE International Conference on Software Testing, Verification and Validation*, pages 200–209, 2011.

[25] J. Wang, S.-K. Kim, and D. Carrington. Verifying metamodel coverage of model transformations. In *Proceedings of the Australian Software Engineering Conference*, pages 270–282, 2006.

[26] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[27] J. Wilhelm and T. Chiueh. A forced sampled execution approach to kernel rootkit identification. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection*, volume 4637 of *LNCS*, pages 219–235, 2007.

[28] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *Proceedings of the 28th International Conference on Software Engineering*, pages 272–281, May 2006.

A Calendar Generator Transform

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output method="xml" encoding="UTF-8"/>
<xsl:variable name="start" select="/month/@start"/>
<xsl:variable name="count" select="/month/@days"/>
<xsl:variable name="total" select="$start + $count - 1"/>
<xsl:variable name="overflow" select="$total mod 7"/>

<xsl:variable name="nelements">
    <xsl:choose>
        <xsl:when test="$overflow > 0"><xsl:value-of select="$total + 7 - $overflow"/></xsl:when>
        <xsl:otherwise><xsl:value-of select="$total"/></xsl:otherwise>
    </xsl:choose>
</xsl:variable>

<xsl:template match="/">
    <html>
    <head><title><xsl:value-of select="month/name"/></title></head>
    <body bgcolor="lightyellow">
    <h1 align="center"><xsl:value-of select="month/name"/></h1>
    <table summary="calendar" border="1" bgcolor="yellow" align="center">
        <tr bgcolor="white">
        <th>Sun</th><th>Mon</th><th>Tue</th><th>Wed</th><th>Thu</th><th>Fri</th><th>Sat</th>
        </tr>
        <xsl:call-template name="month"/>
    </table>
    </body></html>
</xsl:template>

<!-- Called only once for root -->
<!-- Uses recursion with index + 7 for each week -->
<xsl:template name="month">
    <xsl:param name="index" select="1"/>
    <xsl:if test="$index &lt; $nelements">
        <xsl:call-template name="week">
            <xsl:with-param name="index" select="$index"/>
        </xsl:call-template>

        <xsl:call-template name="month">
            <xsl:with-param name="index" select="$index + 7"/>
        </xsl:call-template>
    </xsl:if>
</xsl:template>

<!-- Called repeatedly by month for each week -->
<xsl:template name="week">
    <xsl:param name="index" select="1"/>
    <tr>
        <xsl:call-template name="days">
            <xsl:with-param name="index" select="$index"/>
            <xsl:with-param name="counter" select="$index + 6"/>
        </xsl:call-template>
    </tr>
</xsl:template>

<!-- Called by week -->
<!-- Uses recursion with index + 1 for each day-of-week -->
<xsl:template name="days">
    <xsl:param name="index" select="1"/>
    <xsl:param name="counter" select="1"/>
    <xsl:choose>
        <xsl:when test="$index &lt; $start">
            <td>-</td>
        </xsl:when>
        <xsl:when test="$index - $start + 1 > $count">
            <td>-</td>
        </xsl:when>
        <xsl:when test="$index > $start - 1">
            <td><xsl:value-of select="$index - $start + 1"/></td>
        </xsl:when>
    </xsl:choose>
    <xsl:if test="$counter > $index">
        <xsl:call-template name="days">
            <xsl:with-param name="index" select="$index + 1"/>
            <xsl:with-param name="counter" select="$counter"/>
        </xsl:call-template>
    </xsl:if>
</xsl:template>
</xsl:stylesheet>
```