Automated Support for Repairing Input-Model Faults

Senthil Mani, Vibha Singhal Sinha, Pankaj Dhoolia, Saurabh Sinha
IBM Research – India
{sentmani,vibha.sinha, pdhoolia, saurabhsinha}@in.ibm.com

ABSTRACT
Model transforms are a class of applications that convert a model to another model or text. The inputs to such transforms are often large and complex; therefore, faults in the models that cause a transformation to generate incorrect output can be difficult to identify and fix. In previous work, we presented an approach that uses dynamic tainting to help locate input-model faults. In this paper, we present techniques to assist with repairing input-model faults. Our approach collects runtime information for the failing transformation, and computes repair actions that are targeted toward fixing the immediate cause of the failure. In many cases, these repair actions result in the generation of the correct output. In other cases, the initial fix can be incomplete, with the input model requiring further repairs. To address this, we present a pattern-analysis technique that identifies correct output fragments that are similar to the incorrect fragment and, based on the taint information associated with such fragments, computes additional repair actions. We present the results of empirical studies, conducted using real model transforms, which illustrate the applicability and effectiveness of our approach for repairing different types of faults.

Categories and Subject Descriptors
D.2.5 [Software Engineering]: Testing and Debugging—Debugging aids, Tracing

General Terms
Algorithms, Experimentation, Measurement

Keywords
Model repair, repair actions, execution tracing, pattern analysis, model-driven engineering

1. INTRODUCTION
Model-Driven Engineering (MDE) [13] is the paradigm of software development that uses formal models, at different abstraction levels, to represent the system under development, and uses automated transforms to convert one model to another model or text.\(^1\) A significant class of model transforms, called model-to-text transforms, generate text output (e.g., code, configuration files, or HTML/JSP files) from an input model (e.g., XML or UML). The inputs to the transforms are often large and complex, containing tens of thousands of elements. Therefore, faults in an input model that cause a transformation to fail by generating an incorrect output can be difficult to identify and fix.

Although model-transformation failures can be caused by faults in the transform, the goal of our work is to develop techniques for investigating failures that are caused by faults in the input models. This is particularly relevant in MDE, where it is a common practice for transform users to use transforms that are not written by them, e.g., standard built-in transforms provided by a tool. Therefore, when an execution of such a transform fails, the pertinent task for the user is to locate and fix faults in the failure-inducing input models; investigating the transform code is irrelevant from the user’s perspective.

In recent work [6], we presented an approach for assisting with fault localization—identifying the input-model faults—for model-to-text transforms. That approach uses dynamic tainting [3] to trace the flow of data from the transform input to the transform output. Using the generated taint logs and a location in the output where a missing or incorrect string occurs, the approach identifies the fault space that the user can examine incrementally to locate the fault.

Fault localization is only the first step in debugging an input model; the next step is fault repair: fixing the fault so that the execution of the transform on the fixed model generates the correct output. Although fault localization may identify the problematic parts of the input with high accuracy, fixing the fault may not be obvious. To illustrate, consider a case in which an input-model attribute isGen determines the outcome of a conditional statement in the transform. An incorrect value of isGen may cause a wrong path to be taken and, consequently, an incorrect output to be generated. Even though isGen may be identified precisely during fault localization, fixing the value of isGen may not be obvious. This is especially true when the user has no access to the transform code, which, as mentioned, is often the case. Thus, in addition to fault localization, a technique for supporting fault repair is essential for developing efficient model debugging.

Existing work on fault repair for models (e.g., [8, 9, 14]) considers only those faults that violate consistency constraints; such faults can be detected using consistency checkers. However, in the context of automated transforms, a large class of faults may violate no (model or transform) constraints, and yet cause a transformation

\(^1\) A transform is a function, or a program, that maps one model to another model or text; a transformation is the application, or the execution, of a transform on a model instance [2].
failure. Such faults cannot be detected using automated checkers; therefore, no support for fault repair can be provided.

To extend our work on fault localization [6] and address these limitations, in this paper, we present techniques to support fault repair—fixing the input-model faults. Our approach for fault repair collects runtime information for the failing transformation, and uses the information to compute repair actions to be performed on the failure-inducing input model. For example, a repair action could suggest the addition of an element to the model, or a change in the value of a model attribute.

We define different types of runtime information (or metadata) about the failing transformation that need to be collected to enable the computation of accurate repair actions. The metadata contain information about conditional statements, looping constructs, and how input-model elements and attributes are accessed during the transformation. For example, for a conditional statement, we capture information about the comparison made on an input model element: the model element that is compared, the comparison operator, and the value compared with. To collect the runtime information, we leverage and extend the dynamic-taint analysis developed in our previous work [6].

We present two techniques that leverage the metadata to compute repair actions. The first technique, fault-index analysis, computes repair actions that are targeted toward fixing the immediate cause of the failure. This initial set of repair actions can, in many cases, fix the fault completely so that the transform generates the correct output on the repaired input model. However, in other cases, the application of the repair actions may not generate the correct output because the input model requires further repairs.

To address this, we present a second technique, which performs pattern analysis on the output of the failing transformation to find instances of correct output string fragments that are similar to the incorrect output string fragment. The intuitive idea underlying this technique is that by analyzing correct samples of similar output fragments, appropriate repair actions can be computed. For instance, the correct samples can highlight omissions—missing elements and attributes—that need to be added to the input model. In addition to finding similar fragments in the output of the failing transformation, pattern analysis can be performed on the output generated from exemplar input models.

We implemented our fault-repair approach for XSL-based transforms that convert input models to text (Java code, properties files, and configuration files). Using the implementation, we conducted empirical studies on a set of real transforms to evaluate the usefulness of our approach. Our results indicate that, for the subjects considered, different types of metadata are necessary to enable the computation of accurate repair actions. Moreover, the fault-index-based repair actions suffice for completely fixing many types of faults. For the types of faults for which the initial repair actions are insufficient, the pattern-analysis technique computed meaningful repair actions in each instance.

The main contributions of the work are

- The definition of metadata about failing transformations that enable the computation of accurate fault-repair actions
- The development of two complementary techniques, fault-index analysis and pattern analysis, for supporting repairs of input model faults
- The implementation of the techniques for XSL-based model-to-text transforms
- Empirical results that illustrate the applicability of the techniques in supporting model repairs for real model transforms

2. BACKGROUND

In this section, we introduce an example to illustrate the problem, and discuss our tainting-based fault-localization technique [6].

2.1 Example Transform and Input Model

Model-to-text (M2T) transforms are a class of applications that transform input models to text files. Examples of such transforms include UML-to-Java code generators and XML-to-HTML converters. An M2T transform can be implemented using a general-purpose programming language, such as Java. Such a transform reads content from input files, performs the transformation logic, and writes strings to output files. Alternatively, a transform can be implemented using specialized templating languages, such as XSLT (eXtensible Stylesheet Language Transformation), that let developers code the transform logic as a template. The associated Xalan library (http://xml.apache.org/xalan-j) provides the functionality to read the input and write the output to a text file.

Figure 1(a) presents PrintUser, an example M2T transform written in XSLT. The transform takes as input an XML file containing user information and prints the information in a plain-text format. The XML shown in Part (b) of the figure illustrates a sample input to the transform. The input model contains two <user> elements (lines 2–15 and 16–24), each of which has a <family> element. Each <family> element, in turn, contains a list of <user> elements that specify the spouse and dependents of the containing <user> element. As illustrated, a <user> element can have five attributes: first, last, hasMiddle, middle, and gender. A <user> element also has a <relationship> element contained within it. Each element has a unique node ID, specified as the tag attribute, which is added by our infrastructure to identify input-model elements.

PrintUser iterates over the top-level <user> elements using the XSL <for-each construct> (line 2). Within the loop, it first calls function printUserName (shown in lines 44–52) to print the user name (lines 4–6). Then, it processes <family> elements (lines 7–40): it prints spouse details (lines 8–29) and dependent details (lines 30–39). To print the spouse information, the transform uses the <choose construct> at line 9 to test the gender attribute of <user> (lines 10, 19). If the value of gender is F, the transform selects the contained <user> element whose <relationship> sub-element has type attribute with value husband (line 11) and prints the husband’s name (lines 13–16). Lines 20–26 perform a similar processing for the case where the value of gender is M.

The output of the transform on the sample input is shown in Figure 1(c). Lines 1–9 of the output correspond to the data printed for the first <user> element (lines 2–15 of the input). Transform lines 3–6 print output lines 1–2; transform lines 13–16 print output line 5; and transform lines 30–39 print output lines 7–9.

The input model contains three faults that, which cause incorrect output to be generated. The incorrect parts of the output, which have missing substrings, are highlighted in the output as E1, E2, and E3. E1 illustrates a missing middle name on output line 12. This occurs because of a fault in line 17 of the input: the <user> element is missing attributes hasMiddle and middle. E2 highlights missing text for the spouse name, which occurs because of the fault in input line 21, where element <relationship> is missing attribute type. Finally, E3 highlights missing dependent details; this occurs because element <family> in input line 18 is missing an entire <user> sub-element.

2.2 Fault Localization

In previous work [6], we presented a dynamic-tainting-based approach that can assist in locating such input-model faults. As mentioned earlier, the goal of our work is to assist with locating and
fixing input-model faults (and not transform faults). Therefore, we assume that the transform is correct. Moreover, we assume that the input model satisfies all consistency constraints imposed by the metamodel or required by the transform.

The fault-localization approach associates taints marks with input-model elements and adds probes to the transform that propagate the taint marks to the output string. A key feature of our approach is that it distinguishes different types of taint marks: data taints, control taints, and loop taints, which are propagated at different transform statements. A data taint is propagated at assignments and statements that directly, or indirectly, construct the output string. A control taint is propagated at conditional statements to the parts of the output that are constructed along the branches of the conditional. A loop taint is propagated at looping constructs to the output strings that are constructed within the loops. The taint marks associated with substrings of the output are used to construct a fault space, which the user can examine incrementally to locate the fault.

To illustrate, consider a smaller fragment of the input model of Figure 1(b), shown in a tree form in Figure 2 (left). The model elements and attributes have taint marks, labeled $t_1$ to $t_8$, associated with them. During the execution of the transform, these taint marks are propagated to the output string to generate a taint log. The right side of the figure shows an intuitive representation of the taint log. Different substrings in the output have data, control, and loop taints associated with them. For example, the output string “Jane” has data taint $t_{3,d}$ associated with it because the string is constructed from the input-model attribute `first`, which was initialized with taint $t_3$. For another example, the output string “P.” is generated under a condition (line 47 of `PrintUser`) whose outcome is determined by attribute `hasMiddle`, which was initialized with taint $t_5$; therefore, $t_5$ is propagated to “P” as control taint $t_{5,c}$. A data or a control taint can be empty—i.e., it may enclose no string. This occurs when an input-model attribute is empty or no string is generated along the taken-branch of a conditional whose outcome is determined by an input-model attribute.

Given a taint log and an error marker (i.e., a location in the output text where a missing or incorrect string occurs) identified by the user, the fault-localization technique identifies enclosing taints, in reverse order of nesting, and constructs the fault space incrementally. To illustrate, for failure E1, line 47 of `PrintUser` at attempts to access attribute `hasMiddle` for the second user. Because this attribute does not exist, no control-taint information is generated for the access. Starting at the error marker, our fault-localization technique would search for the enclosing control taints. It would identify the control taint that was propagated at line 2: the loop that lex-

Figure 2 shows an abstract representation of the taint log. In our implementation, the taint log is an XML file in which different elements, and their nesting structure, capture the taint information. We show samples of taint logs in the discussion of our approach (Section 3).
3. AUTOMATED FAULT REPAIR

The goal of our approach for fault repair is to offer recommendations to the user on how to fix a faulty input model that causes a transformation failure. A fix recommendation $R$ consists of a set of repair actions on a faulty input model. A repair action $r$ on an input model is a unit change to be made to the model. We define a repair action as follows:

$$ r ::= \text{ADD attr to elem}_t \quad \text{ADD elem to elem}_t \quad \text{SET attr to value} | -value | * $$

The first action adds an attribute with an arbitrary, possibly empty, value to an element. The second action adds an element to another element. The final action sets an attribute to the specified value, any value other than the specified value, or to an arbitrary value (indicated by ‘*’). $\text{elem}_t$ is an element name, whereas $\text{elem}_r$ can be an element name (for a newly added element by a repair action) or a node ID (for an existing element in the model).

Figure 3 presents the overall approach for debugging input-model faults: it includes both fault localization and fault repair. Given a transform program $T$ and a failure-inducing input model $I_{\text{faulty}}$ for which $T$ generates an incorrect output, fault localization assists in identifying the input-model fault. Given a user-provided error marker and the taint log for the failing transformation, the fault localizer computes the fault space and the fault index $L_f$ in the taint log: the relevant taint mark for a given error marker.

To compute repair actions, we have extended the taint-propagation mechanism to collect additional information (or metadata) about the failing transformation. The fault-repair approach takes as input the fault index in the taint log. It computes a set of alternative recommendations $R$. The user can select a recommendation $R \in R$ and apply the repair actions in $R$ to the faulty input.

Next, we define the metadata required for computing accurate repair actions (Section 3.1). Following that, we present the two fault-repair techniques: fault-index analysis (Section 3.2) and pattern analysis (Section 3.3). Both these techniques leverage a common algorithm for computing repair actions, which we present in Section 3.4. Finally, we present an analytical discussion of the approach (Section 3.5).

3.1 Metadata Collection for Fault Repair

For fault localization, information about the flow of input data to output text in the failing transformation is sufficient. However, to support fault repair, extra information that indicates how the input data was used to create the output is required. This metadata essentially allows us to recreate, to some degree, the transform logic. Our approach collects three types of metadata for the failing transformation: (1) accesses of input-model entities, (2) conditional statements, and (3) selection of model elements from a collection.

Metadata about entity access.

We collect metadata about transform statements that, directly or indirectly, attempt to access input-model entities (elements or attributes). For entities that exist in the input model, this information is naturally captured via taint propagation. However, for a missing entity, no taint initialization occurs; therefore, no taint propagation is performed even though the transform attempts to access the attribute. To handle such cases, we augment the transform instrumentation to collect runtime information about accesses to input-model elements, irrespective of whether the access succeeds.

The metadata is captured in a $\langle \text{get} \rangle$ element in the taint log. This element has two attributes: $\text{xpath}$, which records the path in the input model along which a model entity is accessed, and $\text{startNode}$, which records the input-model element on which the path is executed. Figure 4(a) shows the metadata collected at line 47 of $\text{PrintUser}$ when executed on the faulty input. The $\text{xpath}$ attribute of the $\langle \text{get} \rangle$ element indicates the access of $\text{hasMiddle}$. The $\text{startNode}$ attribute of $\langle \text{get} \rangle$ indicates that $\text{hasMiddle}$ is being accessed on element i.4. Thus, although $\text{hasMiddle}$ for the second $\langle \text{user} \rangle$ element is missing, the collected information indicates that the attribute was attempted to be accessed at line 47.

Metadata about conditional statements.

For a conditional statement (i.e., an $\text{if}$ or a $\text{case}$ statement), our technique collects information about the comparison that occurs at the statement: the left-hand and right-hand expressions (which could represent an input-model entity or a string literal), and the comparison operator. For example, for line 47 of $\text{PrintUser}$, we collect information that attribute $\text{hasMiddle}$ (for a particular $\langle \text{user} \rangle$ element) is compared to “true” using the equality operator.

In the taint log, the information is represented in a $\langle \text{condition} \rangle$ element, with attributes: (1) $\text{startNode}$ and $\text{value}$ to capture the left-hand expression, (2) $\text{op}$ to capture the comparison operator, and (3) $\text{rightNode}$ and $\text{value}$ to capture the right-hand expression. If an expression is a string literal, the node attribute does not occur. If the accessed input-model attribute does not exist, neither the node attribute nor the value attribute exist. The $\langle \text{condition} \rangle$ element in the taint-log fragment for line 47, shown in Figure 4(a), illustrates this. In this case, $\langle \text{condition} \rangle$ is missing $\text{leftNode}$ and $\text{value}$ attributes because the accessed attribute $\text{hasMiddle}$ does not exist in the input model.

Metadata about selection in loops.

Transforms often select specific elements from a collection of elements based on some selection criteria. For such statements, we collect data about the input-model elements being iterated over and the selection criterion being applied to each element. The taint log in Figure 4(b) illustrates the metadata collected at lines 20–21. The $\langle \text{get} \rangle$ sub-element of the $\langle \text{loop} \rangle$ element (which contains information about loop taints) records the selection criterion (an XPath expression) in the $\langle \text{xpath} \rangle$ attribute. XPath captures the fact that the $\text{type}$ attribute of the $\langle \text{relationship} \rangle$ element is compared, using the equality operator, to string “wife.”
3.2 Fault-Index Analysis

The fault-index analysis uses the metadata associated with the fault index to compute repair actions. Consider the taint-log fragment shown in Figure 4(a), in which the fault index for failure E1 is highlighted as $f_1$. The fault index is an empty control taint, which is preceded by a <condition> node. This control taint is propagated during the execution of line 47 of the transform on the faulty input. Thus, the empty control taint indicates that the fault caused the conditional in line 47 of PrintUser to evaluate false. The metadata associated with the <condition> element lets our technique reconstruct the expected evaluation as $leftNode = true$. However, the $leftNode$ attribute is not present in the taint log, suggesting that the input-model entity that was accessed as $leftNode$ did not exist. Next, the <get> element preceding the <condition> element provides information that attribute $hasMiddle$ was accessed on input-model element 1.4. Putting together all this information, the technique computes the following repair actions:

$$ADD\ hasMiddle\ TO\ 1.4 \quad [r_1(E1)]$$
$$SET\ hasMiddle\ TO\ true \quad [r_2(E1)]$$

Figure 4(b) illustrates the fault index for E2, highlighted by the ellipse. Similar to E1, the fault index is an empty control taint. The control taint is preceded by a <next> element whose parent element is a <loop> element. This indicates that the fault causes a loop iteration to not occur. The metadata associated with the <get> sub-element of <loop> captures the criterion that was used to select the elements to be iterated over. Thus, starting at element 1.4, the transform expected the path $family/user/relationship$ to exist in the input such that the $type$ attribute of $relationship$ was $wife$. Moreover, the analysis is able to resolve the access of $relationship$ to ID 1.4.2.2.2 (see line 21 in Figure 1(b)); we discuss the resolution algorithm in Section 3.4. Thus, the fault-index analysis issues two repair actions:

$$ADD\ type\ TO\ 1.4.2.2.2 \quad [r_1(E2)]$$
$$SET\ type\ TO\ wife \quad [r_2(E2)]$$

Part (c) of Figure 4 illustrates the fault index for E3. In this case, the problem is traced to a missing loop iteration, which occurs because the selection criterion is not satisfied. Following a similar approach as for E2, the analysis computes four repair actions:

$$ADD\ user\ TO\ 1.4 \quad [r_1(E3)]$$
$$ADD\ relationship\ TO\ user \quad [r_2(E3)]$$
$$ADD\ type\ TO\ relationship \quad [r_3(E3)]$$
$$SET\ type\ TO\ dependent \quad [r_4(E3)]$$

The repair actions computed by the fault-index analysis target fixing the immediate cause of the failure. However, some faults can require multiple fixes in the input model. For such faults, the repair actions computed by fault-index analysis fix the model only partially. In our example, both E1 and E3 illustrate partial fixes. For E1, after repair actions $r_1(E1)$ and $r_2(E1)$ are applied and the transform is rerun, the desired output (the middle name for user “Amit Vyas”) is still not generated. Although the repair actions fixed attribute $hasMiddle$, attribute $middle$ was still missing in the fixed model. On this execution, the fault index is the empty data taint marked $I_2$ in Figure 4(a). The fault-index analyzer computes the following additional repair actions for E1:

$$ADD\ middle\ TO\ 1.4 \quad [r_3(E1)]$$
$$SET\ middle\ TO\ * \quad [r_4(E1)]$$

In this manner, the user could iteratively perform fault localization and repair and, in each iteration, fix the model partially based on the repair actions computed by the fault-index analysis, until the fix is complete. Clearly, such an approach can be cumbersome. Our pattern-analysis technique, which we discuss next, addresses this problem by identifying all possible repair actions that may be required to generate the correct output.
3.3 Pattern Analysis

Pattern analysis attempts to find output string fragments that are similar to the incorrect string fragment. It does this based on the structure of taint marks in the taint log. Intuitively, the taint marks in the log have a hierarchical tree-like structure. Figure 5 presents an illustrative example. Each taint mark in the log is represented as a node. The edges in the tree represent the nesting among the taint marks. The sibling nodes are ordered, from left to right, based on their sequence of occurrence in the log. Thus, each node has an index associated with it, which specifies the position of the node among its siblings. In the figure, the index of a node is specified as a subscript $\{x\}$. A loop taint has a control-taint child per iteration of the loop, and one control-taint child for the last loop condition evaluation (after which execution exits the loop). In addition to the three types of taints, nodes also represent string literals, which have a subscript $\{x\}$.

The fault index of a taint log maps to a taint node in the taint-log tree. In Figure 5, node $16_{\{1\}}$ is the fault index. Intuitively, starting at the fault index, the pattern analysis traverses up in the tree to identify the (unique) path to the root, which we refer to as the fault path. This path is highlighted in the figure with shaded nodes. Next, the analysis traverses down in the tree, in a depth-first manner guided by the fault path, to find a matching path that is at least as long as the fault path and that ends at a data node. The fault path guides the downward traversal in the following manner. If a loop node is encountered, the traversal explores each child of the node (except the child that occurs in the fault path). At any non-loop node, the traversal continues at the child node whose index matches the index of the corresponding node (i.e., the node at the same depth) in the fault path.

For the fault path $(1, 2_{\{1\}}, 6_{\{3\}}, 11_{\{1\}}, 16_{\{1\}})$, the algorithm traverses node $2_{\{1\}}$—it does not traverse $3_{\{2\}}$ because its index, 2, does not match the index, 1, of the fault node. Because $2_{\{1\}}$ is a loop node, the analysis explores all its child nodes except for $6_{\{3\}}$ because it occurs on the fault path. The analysis traverses four paths, which are emphasized with thick lines in the figure. Of these, the subpaths in dotted lines illustrate the traversals along which the analysis finds no relevant paths for computing repair actions. For example, path $(1, 2_{\{1\}}, 4_{\{1\}}, 9_{\{1\}})$ is irrelevant because its length is less than the length of the fault path. Path $(1, 2_{\{1\}}, 5_{\{2\}}, 10_{\{1\}}, 4_{\{2\}})$ is irrelevant because it does not end at a data or string literal node; Path $(1, 2_{\{1\}}, 5_{\{2\}}, 10_{\{1\}}, 13_{\{1\}}, 17_{\{1\}})$ is also irrelevant of the same reason. The highlighted path $(1, 2_{\{1\}}, 5_{\{2\}}, 10_{\{1\}}, 15_{\{3\}}, 18_{\{1\}}, 19_{\{1\}})$ is the only matching path.

The search for matching paths is guided by the fault path to ensure that a similar string is explored by the pattern analysis. The constraints on the path length (i.e., path must be at least as long as the fault path) and on the last node (i.e., the node must be a data node) ensure that the repair actions computed based on a matching path will lead to a change in the output text at the error marker. The repair actions are generated based on the metadata associated with the taints that are encountered starting at the node where the length of the matching path is equal to the length of the fault path.

To illustrate the pattern analysis, consider the taint log shown in Figure 4(c), which is used to compute repair actions for E3. We illustrate paths using the line numbers in the log. The fault path in the log is $(1, 2, 26, 28, 31, 35)$. To find a similar output instance, the algorithm traverses the matching path $(1, 2, 5, 7, 10, 14, 17)$. In the matching path, it skips the first six nodes, because the length of the fault path is six, and then repair actions are computed based on the metadata information associated with node 17 (i.e., at lines 15 and 16). This part of the taint log is highlighted with a bounding box. The $<next>$ element at line 16 indicates that attribute $first$ of 1.2.2.4 is printed. Generalizing the metadata information to model entity types, we identify that attribute $first$ is required for model entity of type user (1.2.2.4 is a model entity of type user). Therefore, the pattern analysis suggests the following additional repair actions for E3

\begin{algorithm}
\begin{algorithmic}
\State $L$ Taint log
\State $t$ Taint mark in $L$
\State $R$ Fix recommendation for $t$
\Function{ComputeRepairActions}{L, t, R}
\If{$t$ is a data taint}$\langle$set $e$\rangle$\EndIf
\If{$t$'s preceding element $e$ $<next> \land e$ node}$\langle$set $e$.node TO $*$\rangle$\EndIf
\If{$e$'s preceding $<get>$ element $e$ || access of missing entity}$\langle$add $e$.xpath TO $e$.startNode$\rangle$\EndIf
\If{$e$'s preceding $<get>$ element $e$ $<condition>$}$\langle$add $e$.rvalue \land $e$.rvalue $<condition>$\rangle$\EndIf
\If{$e$'s preceding element $e$ $<next>$ then $<miss paths$ iteration}$\langle$find $e$'s $<get>$ element $e$ $<condition>$\rangle$\EndIf
\If{$e$'s preceding element $e$ $<next>$ then $<miss paths$ iteration}$\langle$add $e$'s $<get>$ $e$ $<condition>$\rangle$\EndIf
\If{$e$'s preceding element $e$ $<condition>$ then}$\langle$add $e$.xpath TO $e$.startNode$\rangle$\EndIf
\If{$e$'s precedence element $e$ $<condition>$ then}$\langle$add $e$.xpath TO $e$.startNode$\rangle$\EndIf
\EndFunction
\end{algorithmic}
\end{algorithm}

Combined with the four repairs actions $r_{1-4}$ computed by the fault-index analysis, our approach, thus, computes a recommendation with six repair actions. In general, for each matching path, an alternative recommendation is computed.

3.4 Algorithm for Computing Repair Actions

The fault-index and pattern analyses compute repair actions starting at a taint mark $t$. In the case of fault-index analysis, $t$ is the fault index, whereas in the case of pattern analysis, $t$ is a taint encountered during the traversal of a matching path in the taint tree-structure. Both these analyses leverage a common algorithm that computes repairs actions for a given taint mark. The algorithm consists of three steps.

The first step computes intermediate repair actions. An intermediate repair action can contain an XPath expression. For example, for E2, the intermediate repair actions corresponding to $r_{1-2}$ are computed as $r_{1-2}() = \langle r_{1-2}() \rangle$.

\begin{algorithm}
\begin{algorithmic}
\Function{ComputeIntermediateActions}{L, t, R}
\State $\langle$add $e$.xpath TO $e$.startNode$\rangle$
\State $\langle$add $e$.xpath TO $e$.startNode$\rangle$
\State $\langle$add $e$.xpath TO $e$.startNode$\rangle$
\State $\langle$add $e$.xpath TO $e$.startNode$\rangle$
\EndFunction
\end{algorithmic}
\end{algorithm}

4The “@” symbol in an XPath expression represents access a node attribute.
Figure 6 presents the algorithm that, given a taint mark \( t \), computes repair actions for \( t \). For a data taint, the algorithm checks whether a preceding \(<\text{next}>\) element with a node attribute exists; if it does, the created repair action suggests a change in the value of the named attribute (lines 1–3). If no such element exists, the algorithm looks for a preceding \(<\text{get}>\) element (line 4), which captures information about access of a missing entity; in this case, the algorithm computes two repair actions to suggest the addition of the missing entity with some value (lines 5–6). For a missing loop iteration (lines 8–10), a similar repair action is generated by accessing information about the loop selection criterion (captured in a \(<\text{get}>\) element). For a \(<\text{condition}>\) element, if \( \text{leftNode} \) and \( \text{rvalue} \) exist, a repair action is generated that sets \( \text{leftNode} \) to either \( \text{rvalue} \) or the negation of \( \text{rvalue} \) depending on the comparison operator (lines 12–13). Finally, if \( \text{leftNode} \) does not exist (indicating a comparison with a missing entity) and \( \text{rvalue} \) exists, appropriate ADD and SET actions are created (lines 14–17).

The intermediate repair actions can contain complex XPath expressions, parts of which may already exist in the input model; in the second step, we refine the intermediate repair actions based on such existing parts. This step executes each XPath on the input model to identify parts that already exist in the input. For the non-matching/non-existent XPath parts, we generate repair actions. To illustrate, consider the intermediate repair action for \( E_2 \), which has the complex XPath \( \text{family}/user/\text{relationship}/@\text{type} \). Any of the direct or indirect sub-elements of 1.4 (\(<\text{family}>\), \(<\text{user}>\), \(<\text{relationship}>\)) or the attribute type of \(<\text{relationship}>\) could be missing. We execute this XPath on the input model, starting at 1.4. This returns node 1.4.2.2.2, up to which the match has occurred, but beyond that \( \text{type} \) is missing. Thus, we generate the repair actions \( r_1(E_2) \) and \( r_2(E_2) \), which suggest that a \( \text{type} \) attribute be added to 1.4.2.2.2 and its value be set to \( \text{wife} \).

In the last step, after all the recommendations have been computed, we prune the recommendation set to filter out duplicate recommendations and invalid recommendations. Two recommendations are equal if they consist of the same set of actions. An invalid recommendation is one whose application can alter the output prior to the error marker. The goal of fault repair is to fix the input in such a way that the output at the error marker changes to the expected output; all text preceding the error marker is assumed to be correct. Therefore, we classify a recommendation as invalid if it can potentially change any output text that precedes the error marker.

To illustrate consider the scenario in which the fault for \( E_2 \) has been fixed: i.e., an appropriate \( \text{type} \) attribute has been added to line 21 of the input (Figure 1(b)). In this case, the analysis can recommend two sets of repair actions for \( E_3 \). One of them requires the addition of a new (dependent) \(<\text{user}>\) sub-element to the \(<\text{family}>\) element in line 18 of the input. The second one suggests modifying the \(<\text{user}>\) element in line 19 as follows:

\[
\text{SET} 1.4.2.2.2.\text{type} \text{ TO dependent}
\]

However, because 1.4.2.2.2.\text{type} is used to generate “Spouse Details,” applying this repair action would change the part of output that is assumed to be correct. Therefore, this recommendation is classified as invalid and filtered out.

### 3.5 Discussion

Although we have illustrated our approach using XSLT-based transformations, it is applicable more generally to other transformation implementations, e.g., in Java. For such implementations, the runtime metadata would need to be collected appropriately to generate the taint log. For example, collecting the metadata for loop selections, which is fairly straightforward for XSLT transformations, would require analysis to identify the code that performs the selection. Similarly, identifying the metadata for conditionals would require code analysis and appropriate instrumentation. However, the fault-index and pattern analyses, which operate on the taint log, remain unchanged.

The pattern analysis computes repair actions based on instances of correct strings that are similar to the incorrect string. It presumes that all matching paths identified in the taint tree-structure generate correct strings. Therefore, if a model contains multiple faults, the analysis could use an incorrect string to compute repair actions. In our approach, we assume that all output before the error marker is correct and, if there are multiple faults, the user identifies the error markers in order from the beginning of the output. With these assumptions, the pattern analysis could compute repair actions based on only those strings that precede the error marker; alternatively, it could rank lower the repair actions that are computed based on strings that follow the error marker.

### 4. EMPIRICAL EVALUATION

We implemented our fault-repair approach and conducted empirical studies. The goals of the studies were to investigate (1) the effectiveness of the fault-index analysis, and (2) the characteristics of the recommendation sets computed by the pattern analysis. After describing the experimental setup, we present the results of the two studies.

#### 4.1 Experimental Setup

##### 4.1.1 Subjects and Faulty Inputs

We used six XSL 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, and generates different types of text output: ClsGen1 and ClsGen2 generate Java classes; IntGen1 and IntGen2 generate Java interfaces; PluginGen generates an XML configuration file; and PropGen generates a properties file consisting of name-value pairs. Table 1 lists different XSLT features used in the subjects, which indicates the complexity of the subjects. Columns 2–6 list, respectively, the counts of template (Column 2), for-each (Column 3), if, when, otherwise (Column 4), value-of (Column 5), and variable-param (Column 6). Columns 7–9 present data about the select construct. We classified a select as simple if it contained no condition or axes (e.g., child::), and complex otherwise. As the data indicate, ClsGen2 is the most complex subject, whereas PropGen is the least complex.

To generate failure-inducing input models, we used the technique of generating test inputs by applying mutation operators to a valid input (e.g., [7]). Specifically, we defined four mutation operators: (1) DeleteElement deletes a model element; (2) DeleteAttribute deletes a model attribute; (3) EnumerateAttribute modifies the value of an attribute based on a predefined enumeration of values for the attribute; and (4) EmptyAttribute sets an attribute to empty. We used two valid input models that contain approximately 38,100 and 40,500 entities, respectively. We applied the mutation
operators to the two valid inputs to generate faulty inputs. Given the large sizes of our input models, we used an upper limit of 1000 for each input to bound the number of mutants. From the initial set of mutants, we eliminated equivalent mutants: mutants that caused no change in the transform output.

Table 2 lists the number of faults of different types that were generated for the subjects. Overall, the different types of faults are fairly well-represented; deleted-element faults have a lower representation but, nonetheless, are not insignificant in number. A fair distribution lowers the bias that predominant numbers of some fault types might have introduced in our studies.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Delete Element</th>
<th>Delete Attribute</th>
<th>Empty Attribute</th>
<th>Enumerate Attribute</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClsGen1</td>
<td>33</td>
<td>94</td>
<td>111</td>
<td>33</td>
<td>354</td>
</tr>
<tr>
<td>ClsGen2</td>
<td>23</td>
<td>105</td>
<td>68</td>
<td>249</td>
<td>316</td>
</tr>
<tr>
<td>IntGen1</td>
<td>27</td>
<td>123</td>
<td>56</td>
<td>109</td>
<td>215</td>
</tr>
<tr>
<td>IntGen2</td>
<td>162</td>
<td>344</td>
<td>327</td>
<td>1180</td>
<td>1829</td>
</tr>
<tr>
<td>PluginGen</td>
<td>42</td>
<td>27</td>
<td>29</td>
<td>53</td>
<td>119</td>
</tr>
<tr>
<td>PropGen</td>
<td>8</td>
<td>16</td>
<td>22</td>
<td>95</td>
<td>45</td>
</tr>
<tr>
<td>Total</td>
<td>460 (16%)</td>
<td>804 (27%)</td>
<td>824 (28%)</td>
<td>2949</td>
<td>460</td>
</tr>
</tbody>
</table>

4.1.2 Implementation

We leveraged and extended the implementation infrastructure, which had we developed in our previous work on fault localization for XSLT-based transforms [6]. Our implementation converts an XSLT program to a Java translet, on which all the analyses operate. The infrastructure consists of five main components. (1) A taint API, which contains taint-initialization and taint-propagation methods. (2) An instrumentation component, implemented using BCEL (http://jakarta.apache.org/bcel), which adds probes to the translet bytecode to invoke control-tainting and loop-tainting method. (3) An aspect-weaver component, implemented using AspectJ, which weaves in aspects to the instrumented bytecode to invoke taint-initialization and data-tainting methods. (4) An indexer component, which sanitizes and indexes the raw taint log to make it appropriate for querying. (5) A differencing component, which compares the faulty output with the correct output and identifies the first point at which the output differs (specified as the error marker). Reference [6] provides the implementation details.

As part of the extensions for this work, we bound additional aspects over the translet class, specifically, at calls to iterator and compare methods of the Xalan API to extract metadata information. We implemented the fault-index analyzer and the pattern analyzer in Java to compute all possible recommendations. We automated the execution of all the steps using an Ant build script.

4.2 Effectiveness of Fault-Index Analysis

4.2.1 Goals and Methods

The goals of the first study were to evaluate the applicability and effectiveness of the fault-index analysis. We measure applicability as the percentage of faults for which different types of metadata can be used for computing repair actions. (In the discussion, we refer to the entity-access, condition, and select metadata as $M_e$, $M_c$, and $M_s$, respectively.) Thus, applicability indicates the importance of a particular type of metadata in terms of how often it can be used in practice. We measure effectiveness as the percentage of faults for which the repair actions computed by the fault-index analysis completely fix the faulty model.

To collect the data, we executed each subject on each of its faulty input model. For each execution, we identified the error marker in the incorrect output using a file-differencing utility, and mapped the marker to the fault index in the taint log. Next, we used the fault-index analyzer to compute repair actions. We determined applicability by identifying the mutation operator and the fault index. For example, for operators DeleteElement and DeleteAttribute, if the fault index points to a control taint, $M_e$ and $M_s$ are applicable; if the fault index points to a loop taint, $M_e$ and $M_c$ are applicable; finally, if the fault index points to a data taint, $M_s$ is applicable. Thus, metadata applicability depends on both the type of fault and the structure of the program.

4.2.2 Results and Analysis

Figure 7 presents data about applicability and effectiveness. The chart on the left shows the applicability data. It contains a segmented bar for each subject. The height of a bar represents 100% of the faulty inputs for the subject for which we computed repair actions. Note that these numbers are less than the faulty inputs shown in column 6 of Table 2. This occurs because for some of the faults, no repair actions are required; and no meaningful repair actions can be computed. For example, consider a fault where the value of an attribute name is incorrect, and suppose that name is simply written out by the transform—i.e., it is not used at a condition or a select. In such cases, no suggestion can be made on how the incorrect value of name should be fixed. The fault localization would point the user to name, and it would be up to the user to decide how to change the value.

The segments within the bars represent the percentage of faults for which $M_e$, $M_s$, and $M_c$ are applicable individually. In some cases, (1) $M_e$ and $M_s$ are applicable together, and (2) $M_e$ and $M_c$ are applicable together; the bar chart contains segments for these combinations as well. For example, for ClsGen1, of the total 264 faults, 12 were fixed using the $M_e$ recommendation, 51 were fixed using both $M_e$ and $M_s$, 57 were fixed using both $M_e$ and $M_c$, 66 were fixed using $M_c$, and the remaining 78 required $M_s$.

Overall, the data indicate that $M_e$ is applicable in many cases, typically in combination with $M_c$ and $M_s$. This confirms our belief that $M_e$ collects valuable metadata about accesses of model entities. This information is especially useful when the accessed entity is missing in the model. The percentage of faults for which $M_e$ was required varied from 45% for ClsGen1 to 51% for IntGen2. On average, $M_e$ was required for 46% of the faults. Another observation we make is that $M_e$ is applicable quite frequently as well, both individually and in conjunction with $M_c$. On average, $M_e$ is applicable for 58.1% of the faults.

Figure 7(b) presents the completeness data. In pretty much all cases, for the faulty inputs in which elements were missing (i.e., the inputs created using the DeleteElement mutation operator), the application of repair actions computed by the fault-index analysis resulted in incomplete fixes. In general, repairs performed based on the fault-index analysis would be complete for a missing-element fault only if (2) the element is a leaf element, or (2) none of its attributes or sub-elements are accessed by the transform. We did not observe such cases in our subjects. On average, for 81% of the faults, the metadata-based repairs were complete, whereas for the remaining 19%, the repairs were incomplete.

4.3 Characteristics of Recommendation Sets

4.3.1 Goals and Methods

The goal of this study was to evaluate our approach in terms of the number and sizes of the computed recommendations. For each subject (except ClsGen2) and each faulty input for the subject, we ran the pattern analyzer on the taint logs (with the fault index identified in it) and computed the sizes of the recommendation sets. One
of the steps of our approach is the removal of duplicate recommendations. To assess the benefit of this step, we collected data about the reduction in the number of recommendations after the removal of duplicate recommendations.

4.3.2 Results and Analysis

Figure 8 shows the distribution of the sizes of the recommendation sets. The height of the segmented bar for a subject represents 100% of the faulty inputs for the subject. The segments in a bar depict different ranges of sizes. For example, for IntGen2, for 45% of the faulty inputs, our approach computed recommendations consisting of only one repair action; for 11% of the faulty inputs, it computed recommendations containing two repair actions; for 26% of the inputs, the recommendations contained between three and five repair actions; and for the remaining 18%, the recommendations contained between six and nine repair actions. For PropGen, all the recommendations contained only two repair actions. For none of the subjects, except PluginGen, more than 10 repair actions were computed; the largest recommendation set for PluginGen consisted of 13 repair actions; the average recommendation set size was two. Thus, for our subjects and faulty inputs, the pattern analysis computed recommendations that were not very large.

Table 3 presents data about the reduction achieved by the removal of duplicate recommendations. Maximum reduction occurs when the pruned recommendation set contains only one recommendation; over all subjects, this was achieved for 31% of the faulty inputs. For another 33% of the faulty inputs, at least 95% reduction occurred. For PropGen, after the removal of duplicates, all the recommendation sets contained only one recommendation. On average, 32% of the recommendation sets contained no duplicates; therefore, no reduction occurred for these sets. To summarize, the data indicate that the pattern analysis can often compute duplicate recommendations and that the removal of duplicates can achieve significant reductions. Thus, the removal of duplicates is an important, and effective, step in our approach.

Table 3: Reduction in the sizes of the recommendation sets.

<table>
<thead>
<tr>
<th>Subject</th>
<th>% Recommendation set reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>100%</td>
</tr>
<tr>
<td>ClsGen1</td>
<td>148</td>
</tr>
<tr>
<td>IntGen1</td>
<td>80</td>
</tr>
<tr>
<td>IntGen2</td>
<td>299</td>
</tr>
<tr>
<td>PluginGen</td>
<td>61</td>
</tr>
</tbody>
</table>

4.4 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 six XSLT-based subjects and, therefore, we can draw limited conclusions about how our results might hold for transforms written using other frameworks. However, our transforms vary in complexity and generate different types of outputs (code, properties, and configuration files), which gives us confidence that the results might apply to other XSLT transforms. For transforms written using other frameworks, the effectiveness of the approach would vary depending on the quality of the metadata that is collected.

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. This threat is further mitigated by the fact that different types of faults were well-represented in our set of faulty models. A related threat is that, in our studies, each faulty input contained a single fault. Further experimentation using inputs that contain multiple faults is required to provide evidence of the usefulness of our approach in more general settings.

For this study, we performed the pattern analysis on the taint log for the faulty input model only—we did not use additional exemplar models. Because our input models are large and contain instances of different elements, pattern analysis using only the faulty input worked well. If the faulty input model was not as comprehensive as the inputs used in our studies, or if exemplar models were used to perform the pattern analysis, the effectiveness of pattern analysis might vary. In the extreme case, when no instance of correct pattern is available, only the fault-index analysis (which can compute partial fixes) would be applicable.

5. RELATED WORK

Fault localization and repair is an active research area that has a rich body of work. However, most of the research addresses the problems of locating and fixing faults in programs. Unlike this research, the goal of our work is to provide support for repairing faults in large and complex inputs. From the perspective...
of this goal, existing research that is most closely related to ours is the work on automated repair of model inconsistencies (e.g., [8, 9, 12, 14]) and automated repair of data structures (e.g., [5, 10]). Although the overall goal of this work and our work is similar, the types of faults targeted and the techniques developed for supporting fault repair differ significantly.

Specifically, in the context of UML models, existing research has developed approaches for detecting design inconsistencies in linked models and suggesting repairs to fix the inconsistencies after changes are made to the models [8, 9]. In these approaches, the patterns for design inconsistencies are identified manually; they include both generic patterns that apply to all UML models and model-specific constraints. The fixes could be specified statically along with the pattern, e.g., by using first-order logic formulas [12]. Xiong and colleagues [14] present a language that can be used for writing both consistency relations and fixing behavior. Thus, the existing techniques for supporting model repairs focus on only those faults that can be specified as consistency relations. Our goal is not to fix statically identifiable inconsistencies, but to fix conditions in the input that do not violate consistency constraints, and yet cause a transformation failure.

Existing research in data-structure repair attempts to fix inconsistencies at runtime to prevent unexpected program termination. Demsky and Rinard [5] present a specification-based approach for dynamic detection and repair of data-structure inconsistencies. Elkarableh and colleagues [10] leverage program assertions to identify consistency constraints expected on the input data structures. Their approach uses static analysis and symbolic execution to identify the repair actions at the assertion point. Unlike this research, the goal of our work is to assist the user in fixing faults in input models. Moreover, our work focuses on supporting fault repair after a failure is observed—its goal is not to detect and fix faults on-the-fly during transform execution.

Other techniques for providing automated support for bug fixes focus on faults in programs. Kim and colleagues [11] present an approach for building a catalog of project-specific bug patterns and bug-fix patterns. By analyzing the code changes made to fix bugs, their approach creates a normalized representation of a bug (code that is deleted in a bug fix) and the bug fix (code that is added in a bug fix). Based on pattern matching on such a catalog, their approach can suggest fixes. DebugAdvisor [1] and Hipikat [4] create a project memory by mining relationships between various software artifacts present in different project repositories. For any new bugs, users can query these project memories to understand how similar bugs have been fixed in the past.

6. SUMMARY AND FUTURE WORK

We presented an approach for assisting transform users in fixing faulty input models. We defined different types of information about the failing transformation that are essential for computing accurate repair actions; we presented two techniques that analyze the metadata and compute repair actions. Finally, we demonstrated the feasibility of our approach by implementing it for XSLT transforms.

There are several directions for future work. Although the metadata that our approach requires are generic in nature, the ease with which they can be collected can vary with different transform implementations. Our current implementation has explored the technique for XSLT transforms. Future work could investigate how such metadata can be collected for transforms written using other frameworks or general-purpose languages, such as Java.

To suggest complete fixes for an incorrect output, our approach performs pattern analysis to identify correct output string fragments that are similar to the incorrect or missing output string fragment. To be successful, pattern analysis requires samples of correct output to be available. When such samples are unavailable, which can occur in practical scenarios, pattern analysis would not be applicable. Moreover, a sample input model may not exercise all paths in the transform to generate a particular output. Therefore, the alternative sets of recommendations we compute may be incomplete. To overcome these limitations, predicate switching [15] may be a promising technique, which we plan to investigate.

Finally, user studies are required to evaluate how it can help users locate and fix faults efficiently and effectively.

References