Forensic Debugging of Model Transformations

Mark Hibberd, Michael Lawley and Kerry Raymond

Queensland University of Technology, Brisbane, Australia
mt.hibberd@student.qut.edu.au,{m.lawley,k.raymond}@qut.edu.au

Abstract. Software bugs occur in model-driven development, just as they do with traditional development techniques. We explore the types of bugs that occur in model transformations and identify debugging approaches that can be applied or adapted to a model-driven context. Investigation shows that the detailed source-to-target traceability available with model transformations enables effective post-hoc, or forensic, debugging. Forensic debugging techniques are introduced for automated bug localisation in model transformations. The methods discussed are grounded with examples using the Eclipse Modeling Framework (EMF) and Tefkat, a declarative model transformation engine.

1 Introduction

As model-driven engineering techniques have become widely adopted in commercial environments the need for related high quality, pragmatic engineering processes has become very apparent. A key aspect of this is the need for efficient and effective debugging techniques.

Model transformations form the backbone of model-driven engineering, and correspondingly become a primary point of failure. Transformation development faces similar challenges to traditional programming; specifically, the possibility of human error in any stage of the development life-cycle; thus the need for debugging.

Debugging is readily classified into three parts, identifying the existence of a problem, fault localisation and the actual correction of the problem [1]. Generally, once a problem is located, a developer who is adequately experienced with the technology can correct the problem with minimal effort. Traditionally the majority of effort is spent on bug localisation [1–3] and the case in model transformations is no different. Based on the premise that automation of bug localisation will provide the greatest benefit to the developer or modeler, we describe debugging primarily in terms of this localisation.

In this paper we address bug localisation for model transformations in four key parts: the identification of questions that are asked when debugging model transformations; the classification of model transformation bugs into a set of bug classes and patterns; the exploration of debugging approaches that can be applied or adapted to these types of bugs; and, the demonstration of these approaches to automate bug localisation in model transformations.

2 Concepts and Context

The goal of a model transformation is to produce one or more target models, from an input of one or more source models. When talking about source and target models in the context of a model transformation, there are two parts, the *meta-model* that describes, or defines, the model and the model *instance*, which is a specific occurrence of the meta-model.

2.1 Model Transformation Tools

There are a number of model transformation tools available which utilise different techniques to solve the transformation problem. The defining characteristics [4–6] of these techniques include:

- how the transformation is specified; as a general purpose language or as a problem specific language.
- the transformation approach; either an imperative or declarative approach.
- what needs to be transformed; the types, described by a meta-model or free text; the number of models involved, 1..m source and 1..n target models
- the traceability between transformation artifacts; firstly the detail of the traceability (if any) and secondly the directionality, i.e. can the trace be followed source-to-target, target-to-source or both.
- the level of automation; applied programatically or manually.

To look at debugging problems, the transformation approach and traceability of the transformation are the most important characteristics. It is assumed that most practical transformations will achieve an adequate level of automation and that both source and target models are instances of a well defined meta-model. The number of models involved in the transformation is disregarded as the debugging concepts should extend to any number of models.

Using these characteristics as a guide, we have utilised the Eclipse Modeling Framework (EMF [7] and the Tefkat [8] model transformation engine. Tefkat uses a declarative approach to model transformations. It has a formal trace model, which links the target, source and transformation. Another important feature of Tefkat is that the abstract syntax of a transformation is represented as a model, with corresponding meta-model. This allows the trace model to accurately reference the transformation as well as the source and target models.

Declarative approaches, like Tefkat, concentrate on *what* relationships exist between the source and target, compared with imperative approaches which concentrate on *how* to explicitly transform from the source to target. By defining only the relationships, declarative transformations allow for complete and correct transformations to occur without concern for execution order, source traversal and target creation. The use of a declarative approach does introduce complex concepts that make traditional imperative debugging techniques difficult. The most obtrusive of these is the lack of a pre-defined execution order. This makes interactive or step-through debugging difficult as the execution order is independent from the concrete syntax.

2.2 The Model Transformation Environment

To identify and localise bugs in model transformations, the model transformation environment must be understood. This environment may vary depending on the specific model transformation technologies, however similar concepts are applicable to most types of transformation. The core artifacts in the model transformation environment are the source model(s), the target model(s), the transformation and any available trace information. The Tefkat model transformation environment (figure 1) is comprised of:

- Source extents: One or more source model instances and their meta-models.
- Target extents: One or more target model instances and their meta-models.
- The transformation: The model transformation and its meta-model.
- The trace extent: A trace model instance and meta-model that links target objects to the source objects that contributed to their creation and the transformation rule involved (see figure 2).

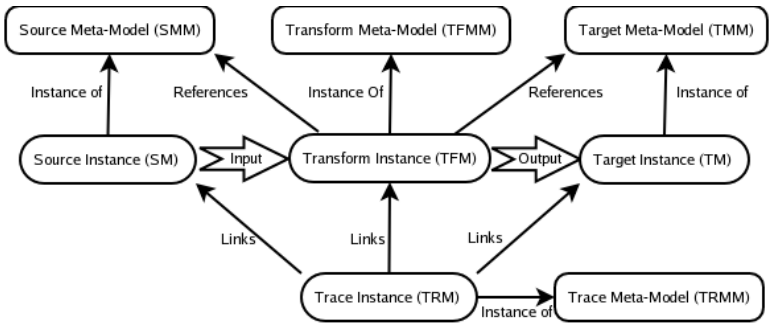

Fig. 1. The model transformation environment.

The trace extent is a key enabling factor for the techniques presented in this paper. Source-to-target traceability was identified as a key requirement in early model research [9–11]. Gerber et al. [12] even noted the possibility of utilising trace information for tasks such as debugging, change propagation and round-trip engineering. As discussed further in section 4, this trace information is yet to be fully exploited for debugging.

The information contained in the trace extent can be leveraged for more effective post-hoc debugging techniques than is possible with traditional languages. Figure 2 provides a visual representation of the information contained by the Tefkat trace model. Each trace object references a target object, the transformation rule which created the target object and the source object(s) which contributed to its creation.

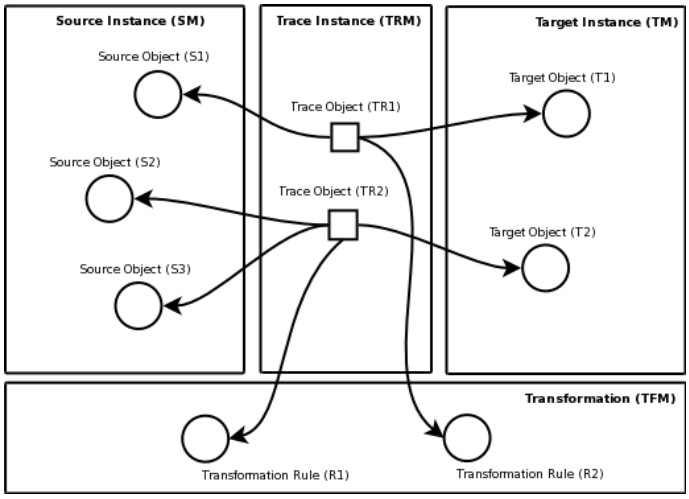

Fig. 2. Model trace environment.

3 Model Transformation Bugs

To start identifying model transformation bugs, the first step is to understand the questions which modelers ask when something goes wrong or just doesn't look correct. These debugging questions, and the resultant classes of bugs are derived from a combination of the author's experience in model transformations and the analysis of transformation problems raised by the Tefkat user community. The identified questions aims to be a complete view of the information required to *identify* and *localise* bugs in model transformations. However, as the questions are based upon experience, it is expected that the set of questions will evolve in the future; as model-transformation techniques are improved and adopted in wider areas of software development.

3.1 The Debugging Questions

The debugging questions commonly asked by a modeler can be divided into two high-level categories. These categories are characterised by model transformations that produce *incorrect* output, logical bugs, compared with those that produce *invalid* output, well-formedness bugs.

Logical bugs, category A, can be identified by the violation of a relationship constraint between the source and target of a given transformation. Commonly these are only informal or implicit constraints, such as, we expect a few x's in the target as we know there are some y's in the source. There are several ways which these constraints could be provided. The first, the constraint could be provided by an oracle, the modeler, as a direct input to the debugging process. Next the constraint could possibly be inferred from the transformation itself. Finally, the constraint could be specified as a part of a formal testing or validation framework.

This paper only deals with the first case, but recognises the benefits of, and the requirement for, more formal specification and/or automated discovery of this type of constraint. The set of logical bug related questions:

- A.1 Why are there no objects of type t in the target?
- A.2 Why are there so many objects of type t in the target?
- A.3 Why is there only one object of type t in the target?
- A.4 Why didn't source type, t, result in any target objects being created?
- A.5 Why doesn't object x contain any references?
- A.6 Why does a particular reference point to object x?
- A.7 Why isn't reference r set?
- A.8 Why are references r_1 to r_n out of order?
- A.9 Why does attribute a have value v?
- A.10 Why isn't attribute a set?

Well-formedness bugs, category B, can be identified by violation of the constraints specified by the target meta-model(s). Handling a model which is invalid with respect to its defining meta-model is a more difficult problem than the incorrect output case. To address this set of questions, debugging tools shall require special case handling and dynamic discovery of the structure of model instances. Dealing with invalid output models is out of scope for this paper, however it is an important direction for future model-transformation debugging research.

The set of well-formedness bugs:

- B.1 Why isn't object x contained?
- B.2 Why was the single valued reference, r, assigned more than once?
- B.3 What violated meta-model constraint c?
- B.4 Why is there no target model at all, i.e. no output compared with empty output as described by question A.1?
- B.5 Why is there an instance, x, of an abstract class c?
- B.6 Why is there an instance, x, that has been created with two different classes c_1 and c_2?

The questions provided for categories A and B are parametrised, indicating the requirement for a debugging context to be well defined and provided as input to the question. The problem of identifying this debugging context results in the definition of a third set of questions, analysis questions. Analysis questions, category C, encompass two sub-groups, *bug smells* or static-analysis questions, category C.I, and information-discovery questions, category C.II. These questions are more about refinement of the problem than debugging questions but they are relevant to localising bugs.

Bug smells represent a pattern or relationship between the source, target and transformation that are commonly the result of a bug. It is important to note that these smells are not always bugs, sometimes there will be a legitimate reason for a bug smell pattern to be found. An example of a bug smells question is question C.I.1.

- C.I.1 Which source objects did not contribute to any target objects?

Bug smell questions often need to be refined to produce more meaningful output. In the example (question C.I.1), there may be a lot of cases where it is acceptable for a source object to not contribute to any target objects. An example of this is where a transformation is not completely exhaustive for the source meta-model. Any objects not referenced by the transformation will not contribute to the target, but is clearly not a bug. This process leads to questions aimed at refining the output. Extending the example in question (question C.I.1).

C.I.2 For all source types, which source objects of the selected type did not contribute to the creation of any target objects?

As these debugging questions evolve, it is apparent that there is a need for supplementary information to use as input to the parametrised debugging questions. This supplementary information is gathered by asking information-discovery questions, category C.II. The information required to answer these questions is often directly available in the trace model, however in large transformations it can still be quite time consuming and error prone to access the information without tool support. The category C.II questions identified are.

C.II.1 Given a target object, what source objects contributed to its creation?
C.II.2 Given a source object, what target objects did it contribute to?
C.II.3 Given an object type, which transformation rules reference the type?
C.II.4 Given a target object, what are the relevant slices of the transformation that could effect its creation and/or attributes?
C.II.5 Which source objects contributed to the creation of target objects?

Figure 3 gives an overview of the debugging question categories that have been defined. The next step in identifying model transformations is to turn these debugging questions into a set of bug categories.

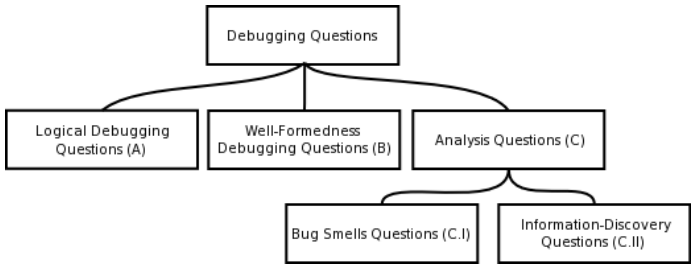

Fig. 3. Question categories.

3.2 Classes of Bugs

The debugging questions raised in section 3.1 allow the classification of possible bugs in model transformations. The bug classes identified can be used to facilitate several decision making processes, including allowing appropriate debugging

approaches to be linked to specific bug scenarios. Following is a set of bug classes with descriptions to identify the different bug scenarios.

Existence bugs: Existence relationships often exist between the source and target, e.g. for all source objects of type x there will be only one target object of type y. These bugs are characterised by the debugging questions A.1-4. Existence relationships are often specified as informal requirements rather than strict rules, which can make them more difficult to identify.

Containment bugs: Meta-models which define containment references expect a strict set of semantics to be adhered to. There are two bugs which result in a containment reference being violated. An object that should be contained is not, or too many objects are contained by a single container.

Bi-directional reference bugs: A common bug pattern with bi-directional references is where both ends of the reference don't point back at each other. Bi-directional constraints are enforced by EMF; this means that a bi-directional reference bug will only result from a bug in the meta-model.

Range bugs: Range bugs occur where there are invalid values in the target instance with respect to the constraints defined by the target meta-model.

Completeness bugs: Completeness bugs occur when some non-optional part of the target is not generated as a part of the transformation.

Well-formedness bugs: Well-formedness bugs are closely related to the category B debugging questions which result from invalid output. A well-formedness bug occurs when the target model instance does not conform to the target meta-model. Well-formedness bugs are a superset of a number of other bug categories, including completeness bugs and containment reference bugs.

Technology specific bugs: Technology specific bugs occur as a result of the model transformation tools and techniques used. This paper limits the discussion of technology specific bugs as there is limited benefit in approaching technology specific problems from a generic model-driven development perspective.

Using these types of bugs as a reference, debugging techniques will now by analyzed to determine effective approaches to bug localisation that may be applied to model transformations.

4 Debugging Techniques

Localisation is the key facet of any debugging process. Figure 4 visualises the goal of bug localisation. The *before* snapshot represents a situation of a buggy transformation, the developer knows there is a bug. However, there is a large area (represented in white) of unexplored code where the bug may be located. The *after* snapshot shows how a debugging process can be applied to narrow the unknown area, and inturn help pinpoint the bugs location. It is important to note that a realistic goal of bug localisation is not to pinpoint the precise problem, merely to localise the possible causes to a minimum area. It is accepted that

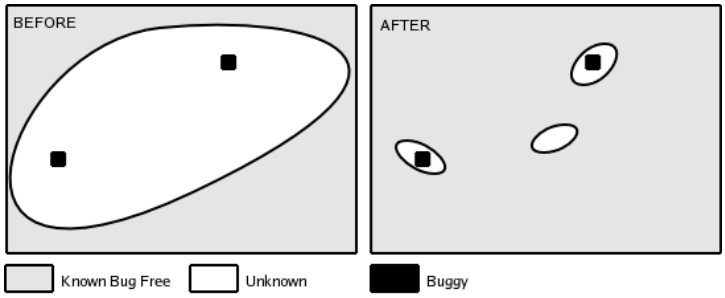

Fig. 4. Before and after localisation of potential problems.

there may always be some level of developer interaction required to go from a localised bug to the correct solution.

To achieve this bug localisation there are two primary categories which can encapsulate most techniques: post-hoc or *forensic* debugging, and interactive or *live* debugging.

4.1 Forensic vs Live Debugging

The key difference between *forensic* and *live* debugging is that live debugging requires access to a complete runtime environment that is not required for forensic debugging. Live debugging may always be required to solve more complex bugs. However in model-driven engineering, the inherent traceability and the well-understood nature of the source and target in model transformations provide a unique opportunity for forensic debugging techniques. Leveraging the additional traceability and artifacts available, a higher level of detail and most importantly automation can be achieved with forensic debugging of model transformations compared with traditional programming languages.

This paper concentrates on a static or *forensic* approach to bug localisation. There are many advantages in pursuing the forensic case over live debugging. These include:

- The time and effort cost for the developer usually associated with interactive debugging.
- In a software development community where automation is continually being pursued, forensic debugging can save valuable resources by using information from failed builds or tests to track down problems rather than relying on developers to re-run the problematic program/transformation to perform live debugging tasks.
- Offline debugging means that bugs can be localised in inaccessible environments, such as production environments where live debugging is often impossible.

– Transient errors or heisenbugs[1] can be difficult or impossible to reproduce in a live debugging environment.

To assist in the development of debugging techniques that can take advantage of these aspects of forensic debugging we look to past debugging research in comparative areas of software development.

4.2 Learning From The Past

There has been no significant research into the post-hoc debugging possibilities specific to model transformation. However, the declarative paradigm used by Tefkat and many other transformation engines is not new to software development. They share similar debugging problems as those seen in fifth generation and logic languages such as Prolog and Mercury. The difficulty in debugging these declarative languages is well-understood, with significant research into debugging techniques such as program slicing [13, 14] and algorithm debugging [15–19].

Many traditional automated debugging techniques such as anomaly detection [20], test based fault localisation [3], statistical based fault localisation [21] and nearest neighbour queries [22] struggle with the paradigm shift from an imperative to declarative approach. However, there is also a portion of automated debugging research which can apply equally, or at least be adapted, to suit both imperative and declarative programs. These approaches include using data-flow analysis to help with program slicing [23], predicate switching [24] and knowledge-based localisation [1].

5 Localisation of Model Transformation Bugs

To address the debugging questions, section 3.1, we present two forensic debugging approaches: analysis and re-enactment. These approaches adapt and extends the techniques discussed above in section 4.2, to best suit forensic debugging of model transformations.

5.1 Analysis

Analysis involves gathering evidence from the artifacts available in the normal model transformation environment (see figure 1). At its simplest level this is simply a collation and refinement of the available data. Analysis is best suited to addressing category C debugging questions by identifying bug smells, and gathering evidence to be used as input to the re-enactment processes. All the information is readily available, however the volume and complexity of the output can prevent viable manual processing.

We have experimented in the use of analysis techniques to gather the information required for bug-localisation. To implement and automate the information gathering required, we have used two different methods. Firstly, programatically

[1] A bug that disappears or changes its behavior when debugging [25].

through the EMF API and secondly, as a Tefkat transformation where the static environment; the original source, target, transform and trace; form the transform inputs and the transform output is a reference or set of references which answer the query. Both techniques have been successful, and the best choice of implementation depends greatly on the specific tools and automation techniques that are being utilised.

The following sections describe, in generic terms, how the model transformation environment can be utilised to answer some of the debugging question using analysis.

Tracing from a target object to its contributers. To address question C.II.1, the required information is readily provided by the trace model (TRM). A direct look-up for each target object will find the rule which created it and the source objects which contributed to its creation.

Tracing from source objects to target objects. Question C.II.2 is effectively addressed by the algorithm specified for question C.II.1, with the source and target roles reversed.

Source objects that contributed to the creation of target objects. The source references in the trace model (TRM), ($TRM[source\text{-}references]$), is a subset of the source model (SM). Using this, question C.II.5 is addressed by finding the intersection of all the source references in the trace model and the objects in the source model.

$$x = TRM[source\text{-}references] \cap SM$$

Source objects that did not contribute to the creation of a target object. Similarity, question C.I.1 is addressed by determining the relative complement of all the source references in the trace model and those in the source model.

$$y = TRM[source\text{-}references] - SM$$

As discussed in section 3.1 and presented in question C.I.2, the output of this question must be refined to produce a useful result. An additional filter is applied to reduce the output; the objects found by the first process that have a type referenced by the transformation (TFM).

$$z = \{o \mid o \in y \wedge o.class \in TFM[MOFInstance]\}$$

Analysis of the model transformation environment has provided enough information to answer straight forward, query based questions. Re-enactment extends this information to pinpoint specific rules and/or terms in rules that trigger a bug.

5.2 Re-enactment

Re-enactment involves the selective re-execution of logical parts of the model transformation in a controlled runtime environment to gather knowledge about specific problems. Typically there are two parts to the re-enactment, determining a part of the transformation which could potentially cause a given problem, and executing that part in isolation. The execution phase of the re-enactment will utilise program slicing [13, 14] and predicate switching [24] to narrow down possible failures over a number of iterations.

The re-enactment process developed involves executing modified slices of the transformation in isolation. The transformation slices shall be created using predicate switching to replace irrelevant or at least suspected irrelevant parts of the transformation. The predicate switching is implemented by replacing conditional terms with an explicit `TRUE` term. The re-enactment algorithms have been designed with automation in mind. As such, they utilise only information available in the model-transformation environment and they do not rely on any additional knowledge to be provided by the user.

Re-enactment is best suited to answering category A debugging questions. For the following examples it is assumed that the output is always valid, but is not the expected output.

Choosing a slice. The process of choosing a slice is dependent on the constraint being tested. Currently this research assumes that a slice has already been identified. There is space for future work in this area, as it may be possible to choose a slice using a heuristics based approach for selecting rules from the transformation or by interacting with a test framework that uses formal constraints to identify bugs.

An example. The first example, figure 5, shows a Tefkat transformation rule.

```
RULE FindPersistentClasses
  FORALL  UMLClass uml
  WHERE   uml.kind = "persistent" AND uml.parent.kind = "persistent"
  MAKE    UMLClass result
  SET     result.name   = uml.name, result.kind   = uml.kind,
          result.parent = uml.parent;
```

Fig. 5. Simple Tefkat rule containing a bug.

This rule is working with a simplified UML meta-model, figure 6. The rule is attempting to locate all persistent classes, that is classes with a `kind` attribute of "persistent" or a parent with a `kind` attribute of "persistent".

The rule contains a bug in the conditional logic. The use of `AND` instead of `OR` prevents the finding any persistent classes. The modeler knows that there are some persistent classes in his input model, so he asks debugging question

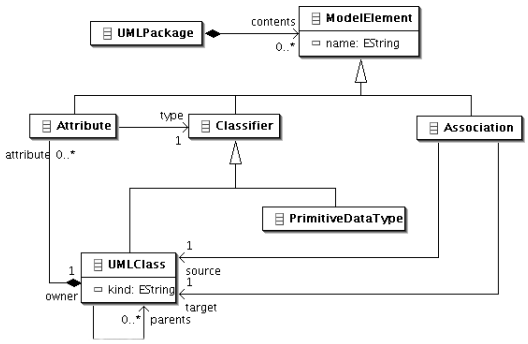

Fig. 6. Simple UML meta-model.

A.1,"why are there no *UMLClasses* in my output?". To answer this question re-enactment is used.

A simple slice. To answer the question posed, a *head-first* or *tail-first* predicate switching approach can be applied to this problem. Our experiments have identified benefits to both approaches. An important point to note when evaluating each approach is that the *head* or *tail* is logical only, to re-iterate the point in section 2, there is no explicit execution order so the terms in the rule may be re-ordered by the transformation engine as required. The choice of starting from the *head* or *tail* is arbitrary, but draws on techniques commonly applied by developers attempting to localise a bug.

The head-first switching algorithm is shown in figure 7. This approach replaces all conditional terms with a `TRUE` term, adding the conditions back one at a time until the transformation output goes from the *correct* output to the *buggy* output. The last term switched is identified as a potential problem. If no terms made a difference to the output it indicates that the input to the rule actually caused it to produce the unexpected output.

The tail-first switching algorithm is shown in figure 8. Tail-first switching iterates through each conditional term, replacing it with a `TRUE` term until the transformation goes from the "buggy" output to the "correct" output. Similar to the head-first approach, if no terms made a difference to the output it indicates that the input to the rule actually caused it to produce the unexpected output.

To produce a complete picture it is possible to combine both approaches. Often both approaches will return the same result, but it is possible that two potential problems can be identified. An advantage of using both approaches is the removing terms in different orders helps the elimination of *down-stream* bugs; those which would not occur except for a problem earlier on in the rule.

In the uml example, applying both of these rules highlights the `uml.parent .kind = "persistent"` term. As highlighted by figure 4 and section 5 this may not be the root cause of the bug, however it localises the problem sufficiently to

```
1  Select source term for rule (current slice)
2  mofInstances <- new list, conditions  <- new list
3  For each term
   3.1  If source term is a MOFInstance, add to mofInsances
   3.2  Otherwise term is potentially a condition, add to conditions
4  For each condition in conditions
   4.1  Replace condition with TRUE term
5  Execute new version of rule
6  If result contains NO required objects, the input is at fault,
   return mofInstances as potential bug
7  For each condition in conditions (from head to tail)
   7.1  Replace TRUE term with condition
   7.2  Execute new version of rule
   7.3  If result contains NO required objects, return condition
8  return Rule is OK
```

Fig. 7. Head-first predicate switching.

```
1  Select source term for rule (current slice)
2  mofInstances <- new list,  conditions <- new list
3  For each term
   3.1  If source term is a MOFInstance, add to mofInsances
   3.2  Otherwise term is potentially a condition, add to conditions
4  For each condition in conditions (from tail to head)
   4.1  Replace condition with TRUE
   4.2  Execute new version of rule
   4.3  If result contains any of required object, return condition
5  No terms effected output, the input is at fault,
   return mofInstances as potential bug
```

Fig. 8. Tail-first predicate switching.

realise that it doesn't make sense that the term always has to be true and the bug can be corrected by modifying the `AND` to an `OR`.

There are a number of caveats to this approach. Most importantly, it is not possible to easily differentiate between a source term that will bind a variable and one that acts as a condition or filter. This means that removing the source term could break the injection part of the rule and cause the transformation to flounder[2]. To address this problem the transformation rule can be modified to not depend on any variables possibly bound in the source term. The first step to this is eliminating all target conditions (`SET` clause in the example). The second step is to eliminate all non-default injections. The example does not have any non-default injections, which take the form of a `MAKE ...FROM ...` clause. These changes to the transformation rule do not affect the algorithms in figures 7 and 8 as, although some values will differ from the "correct" output, there will be no changes to the objects that are created.

An advanced slice. The algorithms presented in figures 7 and 8 address a simple case where all the effecting logic is encapsulated within a single rule

[2] The transformation can not complete execution as a rule is dependent on variable that is never bound.

and with no branching. A more realistic example would involve the use of an `OR` condition, `IF/THEN/ELSE` statement, `PATTERN` use or implicit dependencies between rules created by `LINKS/LINKING` terms. These more complex structures require additional checks that must be made to ensure a complete set of results is determined. For example, in the case of `OR`, a potential problem could be identified for each branch within the rule.

Figure 9 shows the recursive execution of the predicate tail first switching algorithm on each branch of the `OR` condition. This algorithm can be inserted at step 7.1 in figure 7 or step 4.1 in figure 8. The first additional check is for an `OR` condition. If either of these statements are encountered, each of its branches must be traversed separately. It is possible that the conditional statement will result in 0, 1 or 2 additional results.

```
1.  If the condition is an 'OR' term
    1.1  Replace the first nested term with FALSE and recursive apply
         predicate switching algorithm to right hand side of 'OR' term
    1.2  Replace the second nested term with FALSE and recursive apply
         predicate switching algorithm to left hand side of 'OR' term
    1.3  Replace entire 'OR' term with TRUE
2.  Otherwise continue normal predicate switching algorithm
```

Fig. 9. Handling multiple branches.

Other advanced constructs; `IF/THEN/ELSE` statements, `PATTERN`s and `LINKS/LINKING` statements; can be approached with similar predicate switching algorithms. The `IF/THEN/ELSE` case is identical to the `OR` case where each branch is replaced then the whole statement is replaced. Patterns can be addressed by identifying and recursively applying the predicate switching to each `PATTERN` declaration, and finally to the `PATTERN` use. This approach can be used to identify any terms inside the `PATTERN` declaration that effect the output.

Dependencies between rules, normally identified by the `LINKS` construct in Tefkat, present additional problems. In the simple slice example it was noted that floundering could be prevented by modifying the `MAKE` and `SET` clauses to only depend on variables that are not bound by the terms involved in the predicate switching. A rule containing a `LINKS` term may not depend on any other input and as a result the `LINKS` term can not be switched out. There are two approaches to handling this situation. Firstly, the `LINKS` term can be processed last (similar to the MOFInstances in the simple slice). If none of the other terms affect the output then it can be said the rule does not produce the expected output as no dependent objects were created. This information can be used to identify the rules which create those dependent objects, allowing the debugging questions to be asked again for the new rule. The second approach is to ensure that the dependency always exists. This approach is useful when there is more than one `LINKS` term that must be processed.

6 Conclusion

The key to addressing the debugging problem, with respect to model transformations, is understanding the types of questions raised when a problem is identified. In section 3.1, we presented a framework, as a set of questions, to define the goals of model-tranformation debugging.

Utilising forensic debugging approaches we have addressed a number of the model transformation debugging questions highlighted. We have demonstrated the potential that leveraging the trace available in model transformations brings to forensic debugging. We have also demonstrated the adaptability of previously live debugging approaches into forensic algorithms.

Analysis techniques do benefit from leveraging the current trace information. However as the research has progressed it has highlighted the potential for improvements to the information provided by the trace model. Some of these possible enhancements include linking target objects to the specific *injection* that created them and also the rules that resulted in the objects' attributes being set.

The re-enactment approach is able to greatly extend the value which forensic debugging can provide. However, it is important to realise that the re-enactments can rarely (if ever) provide a definitive answer to its queries without help from the user. That said, it contributes significantly towards solving the original problem of localising the fault and minimising developer debugging effort.

6.1 Future Work

In presenting these debugging approaches we have found that some of the debugging questions lend themselves to a forensic debugging solution more than others. In some cases this can be attributed to the level of detail provided by the trace model which is insufficient to answer state based questions, requiring closer examination of the execution chain and intermediate states. In the future we aim to propose improvements to the current trace model and extend our debugging algorithms into the live debugging space to assist in answering more complex and state based debugging questions.

Section 3.1 identified a category of questions to do with the well-formedness of the output; category B. This set of questions have not been thoroughly addressed due to the complexities in handling an *invalid* model instance. We aim to investigate this special case of debugging question further in the future.

Another important piece of work was highlighted in section 3.1. Category A questions were based around constraints on the relationship between the source and target models. It was noted that these constraints need to be communicated more effectively to assist in automating the debugging process, Currently the only way this type of constraint is provided is through manual interaction between the developer and the debugging tools.

References

1. Sedlmeyer, R., Thompson, W., Johnson, P.: Knowledge-based fault localization in debugging: preliminary draft. Proceedings of the ACM SIGSOFT/SIGPLAN software engineering symposium on high-level debugging **8**(4) (1983) 25–31
2. Ducassé, M., Emde, A.: A review of automated debugging systems: knowledge, strategies and techniques. Proceedings of the 10th international conference on Software engineering (1988) 162–171
3. Jones, J., Harrold, M., Stasko, J.: Visualization of test information to assist fault localization. Proceedings of the 24th international conference on Software engineering (2002) 467–477
4. Mens, T., Czarnecki, K., Van Gorp, P.: A Taxonomy of Model Transformation. International Workshop on Graph and Model Transformation (2005)
5. Czarnecki, K., Helsen, S.: Classification of Model Transformation Approaches. Proceedings of the 2nd Object-Oriented Programming, Systems, Languages and Applications (OOPSLA) Workshop on Generative Techniques in the Context of the Model Driven Architecture (2003)
6. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal **45**(3) (2006) 622
7. Eclipse Foundation: Eclipse Modeling Framework Project (EMF). Available at: http://www.eclipse.org/modeling/emf/ (2007) accessed February 20th, 2007.
8. Lawley, M., Steel, J.: Practical Declarative Model Transformation With Tefkat. Lecture Notes In Computer Science **3844** (2006) 139
9. Object Management Group: MOF 2.0 Query - Views - Transformations RFP. OMG Document ad/2002-04-10, April (2002)
10. Miller, J., Mukerji, J., et al.: MDA Guide Version 1.0.1. OMG Document omg/2003-06-01, June (2003)
11. DSTC-IBM-CBOP: MOF 2.0 Query/Views/Transformations, Second revised submission. OMG Document ad/2004-01-06, January (2004)
12. Gerber, A., Lawley, M., Raymond, K., Steel, J., Wood, A.: Transformation: The Missing Link of MDA. Lecture Notes in Computer Science **2505** (2002)
13. Weiser, M.: Programmers use slicing when debugging. Communications of the ACM **25**(7) (1982) 446–452
14. Xu, B., Qian, J., Zhang, X., Wu, Z., Chen, L.: A brief survey of program slicing. ACM SIGSOFT Software Engineering Notes **30**(2) (2005) 1–36
15. Shapiro, E.: Algorithmic program diagnosis. Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (1982) 299–308
16. Shapiro, E.: Algorithmic Program DeBugging. MIT Press Cambridge, MA, USA (1983)
17. Fritzson, P., Shahmehri, N., Kamkar, M., Gyimothy, T.: Generalized algorithmic debugging and testing. ACM Letters on Programming Languages and Systems (LOPLAS) **1**(4) (1992) 303–322
18. Naish, L.: Declarative Debugging of Lazy Functional Programs. Dept. of Computer Science, University of Melbourne (1992)
19. Naish, L.: A Declarative Debugging Scheme. Department of Computer Science, University of Melbourne (1995)
20. Hangal, S., Lam, M.: Tracking down software bugs using automatic anomaly detection. Proceedings of the 24th International Conference on Software Engineering (2002) 291–301

21. Liu, C., Yan, X., Fei, L., Han, J., Midkiff, S.: SOBER: Statistical Model-based Bug Localization. Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (2005) 286–295
22. Renieres, M., Reiss, S.: Fault localization with nearest neighbor queries. Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on (2003) 30–39
23. Agrawal, H., Horgan, J., London, S., Wong, W.: Fault localization using execution slices and dataflow tests. Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on (1995) 143–151
24. Zhang, X., Gupta, N., Gupta, R.: Locating faults through automated predicate switching. International Conference on Software Engineering (2006) 272–281
25. Bourne, S.: A conversation with Bruce Lindsay. Queue **2**(8) (2004) 22–33